The Pragmatic Programmers

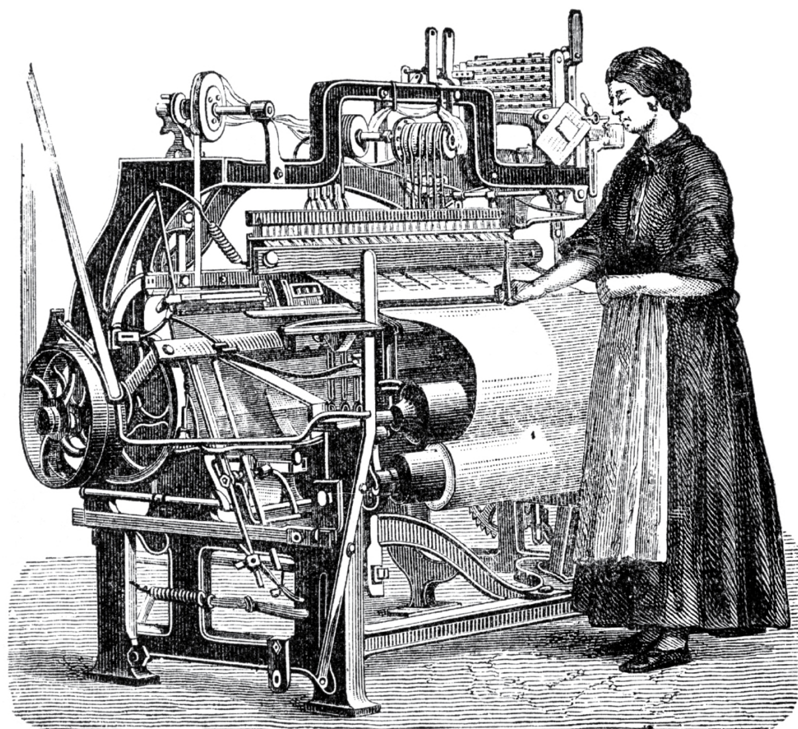
TURING

Driving Technical Change

Why People on Your Team Don't Act on Good Ideas, and How to Convince Them They Should

布道之道

引领团队拥抱技术创新



[美] Terrence Ryan 著

李缨 李松峰 译



人民邮电出版社
POSTS & TELECOM PRESS

“在这本书里，Terrence Ryan清晰地分析和概括了常见但又难以捉摸的人性——怀疑。同时，他也针对推动循序渐进的改变给出了切实可行的解决方案。人类确实经常会莫名其妙地抗拒遵循最佳实践，但这本书不仅有助于认清人们这么做的深层原因，还会告诉你如何在逆境中成功。从本质上说，这是一本讲模式的出色书籍。”

——Ben Nadel，Epicenter 咨询公司首席软件工程师

“人际关系是让搞技术的人最头疼的一个话题，也是技术图书中很少涉及的一个话题。Terrence在本书中颇有见地而又条理分明地对这个话题进行了剖析。本书可以帮你理解各种反对者类型，帮你有策略地把人们引导到你的技术方向上来。”

——Bill Karwin，《SQL反模式》作者

“Ryan以工程师的眼光、心理咨询师的洞见、一线战士的经验，为读者提供了一整套解决最迫切问题的系统方案。”

——Jeff Porten，国际咨询顾问

“这是第一本讲述如何推广技术成果的书，它回答了从事应用或Web开发的程序员都关心的问题。Terrence Ryan的语言幽默风趣，丰富的实例让人身临其境。读罢恍如作者刚刚在你们公司会议室里做完演示一样。”

——Brian Rinaldi，Adobe公司Web社区经理

图书在版编目（CIP）数据

布道之道 : 引领团队拥抱技术创新 / (美) 瑞恩
(Ryan,T.) 著 ; 李缨, 李松峰译. -- 北京 : 人民邮电
出版社, 2012.1
书名原文: Driving Technical Change: Why People
on Your Team Don’t Act on Good Ideas,and How to
Convince Them They Should
ISBN 978-7-115-26727-6

Ⅰ. ①布… Ⅱ. ①瑞… ②李… ③李… Ⅲ. ①领导学
—通俗读物 Ⅳ. ①C933-49

中国版本图书馆CIP数据核字(2011)第226206号

内 容 提 要

本书旨在告诉读者如何说服自己的同事采用新的工具和技术。作者总结了7种怀疑论者模式：孤陋寡闻型、随波逐流型、百般挑剔型、激情燃尽型、时间紧迫型、发号施令型和不可理喻型。围绕说服这些怀疑论者，书中介绍了9个实用技巧和4个策略。理解并熟练运用这些技巧和策略，能够让你的技术布道生涯收获累累硕果。

本书适合IT行业的技术布道师、推广专家、产品经理、需求调研及实施人员阅读，同样也适合对前沿技术时刻保持深厚兴趣的设计和开发人员参考。

布道之道：引领团队拥抱技术创新

◆ 著　[美] Terrence Ryan
译　李　缨　李松峰
责任编辑　朱　巍

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷

◆ 开本：700×1000　1/16
印张：9.25
字数：133千字　　2012年1月第1版
印数：1－3 500册　　2012年1月北京第1次印刷

著作权合同登记号　图字：01-2011-6292号

ISBN 978-7-115-26727-6

定价：29.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

本书赞誉

从本质上说，这是一本讲设计模式的出色书籍。在这本书里，Terrence Ryan对常见但又难以捉摸的人性——怀疑，清晰地进行了概括。同时，他也针对推动循序渐进的改变给出了切实可行的解决方案。人类确实经常会莫名其妙地抗拒遵循最佳实践，但Terry的这本书不仅有助于认清人们抵制改变的原因，还会告诉你如何在逆境中成功。

——Ben Nadel，Epicenter咨询公司首席软件工程师

人际关系是让技术人员最头疼的一个话题，也是技术图书中很少涉及的一个话题。Terrence在本书中颇有见地而又有条有理地对这个话题进行了剖析。本书可以帮你理解各种反对者类型（既有理性的也有非理性的），帮你有策略地把人们引导到你的技术方向上来。

——Bill Karwin，《SQL反模式》作者

Ryan以工程师的眼光、心理咨询师的洞见、一线战士的经验，为读者提供了一整套解决最迫切问题的系统方案，以及如何在技术工作中变得更有创造力和更少挫折感的大局观。本书完全贴近最需要学习Ryan成功的组织管理的读者的切身感受。

——Jeff Porten，国际咨询顾问

这是第一本讲述如何推广技术成果的书，它回答了从事应用或Web开发的程序员都关心的问题。Terrency Ryan的语言风格幽默风趣，丰富的实例让人身临其境。读罢恍如作者刚刚在你们公司会议室里做完演示一样。

——Brian Rinaldi，Adobe公司Web社区经理

致　谢

没有Dave Thomas和Andy Hunt，就没有这本书。如果不是读过《程序员修炼之道》[HT00]，我不可能知道自己迫切需求的技术和工具。从看他们写的书到为他们的出版公司写书，我完成了一个自己曾经认为不可能完成的任务。

Jackie Carter是这本书的编辑。她对我的情况可以说是了如指掌，每当我写不下去的时候，她总会及时地出现。她还给我出谋划策，确保了这本书的可读性。还要感谢本书的技术审校，感谢他们对本书的润色和修饰：Rachel Davies、Ben Nadel、Karl W. Pfalzer、Craig Riecke、Johanna Rothman和Brian Rinaldi。

我必须感谢沃顿商学院的所有同事。无论是质疑过我的人，还是被我说服的人，他们教会了我很多，特别是在做技术布道的时候，该做什么，不该做什么。其中，特别要感谢Dave Siedell、Bob Zarazowski、Gerry McCartney和Deirdre Woods，感谢他们哪怕是心里并不服气，但仍然宽宏大度，一贯支持。

我还要感谢目前Adobe公司的同事。我们这个组都是做技术推广工作的。我们的第一次内部会议考验的是每个人的音量。我特别想感谢我的顶头上司Kevin Hoyt，感谢他提出的中肯建议。Ryan Stewart则是灵感、斗志和鼓舞的源泉。Adam Lehman对我帮助很大，也是和我斗嘴的好搭档。Ben Forta是我最初的榜样，他让我知道自己必须站出来说服大家去争取更大更好的成绩。

Avish Parashar是我身边的老师，他告诉我怎么讲话、站立以及倾听。他教会了我先去做了再说。这本书就是这种思维方式的产物，如果没有他，就不会有这本书。

老爸、老妈，还有Casey，谢谢你们撑起了这个家，在这个家里，通过学习提升自己是一种高尚的行为。Jack还有Ellie，给你们换尿布、喂奶，哄你们笑，让我觉得这辈子都值得了。

最后，我要把最重量级的感谢送给我的贤妻Janice。你从来都不怀疑我能做任何事。你对我的这种信心感染了我。谢谢你相信我可以做到。如果不是为了你，我还真做不到。

目　　录

第一部分　导　　言

第二部分　怀疑者模式





第三部分 技 巧







第四部分 策 略



Part 1 第一部分

导　言

本部分内容

第 1 章

为什么写这本书

我曾经在一家非常有代表性的公司上班，时不时和一群非常有代表性的人在一起，开那么一次非常有代表性的会议。我掌管着公司的Web应用服务器。工作职责包括维护软硬件、推行最佳实践，说服人们不断做技术升级。我一直都忙着想办法让人们升级。

事实上，会议的主题就是升级。

“我想让你们给我一个从ColdFusion 6迁移到ColdFusion 8的时间表。”我说。这已经是第五次在类似会议上给他们下最后通牒了。

不出所料，有人叹了口气说：“我们不能迁移到新服务器。每次把应用迁移到新服务器都会遇到麻烦和不兼容的问题。我们就是不想再给用户添麻烦了。”

我还击道：“这个问题很正常。你们想过在切换版本的过程中，使用单元测试来验证新版本的优势吗？并不是只有应用服务器存在这个问题，Web服务器、数据库服务器，还有你们的基础代码都存在这个问题。版本升级是谁也逃避不了的。你们可以使用单元测试嘛。”

然后，各种借口都涌过来了，“升级会占用太多时间。”“为什么这件事非做不可呢？”“我们不知道怎么做单元测试。”

我可以告诉你，我是在跟他们辩论。我还可以把辩论的情形描述得更详细一些。但那又有什么用呢？结果是明摆着的，我没有把他们争取过来，我始终无法说服他们。

在从事那项工作的时间里，我得想方设法地说服人们在某方面加以改进



或采用某种新技术。刚才描述的争论场面以及类似的情景，我经历了很多很多次了。我输掉了大部分争论，也赢了一些，有些争论到最后终于两败俱伤。不过，我倒是总结出了一些模式：

- 同一类人所持观点相同；
- 有些人始终都乐于接受新事物；
- 另一些人则在别人转变之后才肯改进；
- 有些人你永远也说服不了；
- 某些观点对有些人有用对另一些人没用；
- 有时候让管理层插手是让人们服从的唯一方法。

总结出以上模式之后，我又据以做了很多笔记，明白了某些策略只对部分人有用，永远不能一概而论。然后，我基于不同问题针对同一批怀疑者采取了相同的战术。成功率节节攀升，说服人们有所改进变得简单了。



本书的主要内容就是这些改进、模式，以及争论。但愿我的经验能成为读者的前车之鉴，让大家能够避免不会有结果的争吵，减少挫折感，真正推动组织的技术进步。

1.1 本书组织方式

这是一本讲模式的书。换句话说，本书的主题将围绕一系列重复的形式，也就是模式展开。本书两个主要部分各自包含一组模式，其中一部分模式适用于怀疑者，着重分析某些人抵制的原因。另一部分主要谈方法论，包含一套可以用来反击怀疑者的方法和技巧。如此一来，本书这两部分中各章的结构自然也就极为明晰了。

除了讨论你的同事以及如何对付他们的有关模式的这两部分之外，本书最后一部分将把焦点从模式转移到策略。策略能排序出要先说服谁，不要招惹谁，以及如何真正推动技术进步。

1.2 为什么要看这本书

本书的目标是使你能够说服自己的同事采用新工具和新技术。为了达到

这个目的，你不必像政客一样在办公室里玩阴谋诡计。当然，我可不是说不需要讲策略，只不过用不着那么邪恶罢了。

这本书要牵扯到一大批人物，其中一些人会让你觉得似曾相识——哎，这不就是那谁吗！好，只要你能让现实当中的某个人与书中的人物对号入座，就可以在本书中找到对付他们的方法。对那些怀疑者使用这些方法应该讲究策略，这些本书都会向你一一道来。然后，变化就会魔法般地发生了。

好吧，哪能靠什么魔法啊！你还得靠自己的工作和努力。只不过现在你的目标明确了。看完这本书，相信你一定会有相见恨晚的感觉。而只要你依照本书的策略行事，随着经验的增长，更多的好处还在后面等着你呢。

1.3 本书目标读者

本书的目标读者是搞技术的人，比如开发人员或者程序员。具体来说，可以是服务器管理员、网络工程师或硬件工程师，抑或数据库管理员，甚至是与技术人员协同工作的设计师。

至于你到底搞什么技术，其实并不重要。只要你掌握某种技术，需要与别人沟通，这本书都适合你看。

本书要讲的故事和描述的情景主要涉及开发人员。非常抱歉，我本人就是一名开发人员。但实际上，无论你使用什么工具、什么语言，都没有问题。说到底，不管你是.NET程序员、搞Java开发的，还是开源的拥趸，或者对某家公司的技术情有独钟，都不影响你看这本书。这本书要讲的就是怎么让你的同事改变他们的工作方式，至于改变成哪种方式，那是你自己的事。



第 2 章

开宗明义

本书讨论如何向怀疑者推销业内发展成果。在讨论这个问题之前，我想还是有必要解释一下前面这句话，为此要回答以下问题。

- 什么是业内发展成果？
- 怀疑者指的是谁？
- 为什么需要推销？

2.1 什么是业内发展成果

业内发展成果包括各种工具和技术，使用这些工具和技术可以让你的开发更富有成效，让你的产品更不容易蒙受损失，让你的代码更容易被自己的搭档理解。这个定义涵盖面比较广，因此还需要具体解释一下。

说到提高工作效率，人们往往会首先想到自动化和代码生成，这两种技术都能让你在相对短的时间内产出更多代码。实际上，语言本身也有效率的差异。如果换一种语言就可以用更少的代码实现更多功能，那这也算。虽然可能会引发争议，但我个人还是认为：选一款得心应手的操作系统，也将提高你的工作效率。

至于降低受损害的风险，恐怕头等大事就要数源代码管理了。今天，要是没有进行某种形式的源代码管理，你就不会有安全感。可光有源代码管理还不够。你还得通过单元测试来降低Bug的威胁，就像做UI测试一样。代码评审也可以提高安全系数。总之，只要是能让你晚上睡觉睡得更安稳，或者说即使哪个人意外离职也不会影响开发正常进行的技术，都算是可以降低损害的。



最后，我们经常忽视让团队成员之间更好地沟通的价值。想一想，那些关于写注释、用制表键，还有如何命名变量的争论，归根结底都是缺乏沟通的表现。让其他开发人员轻松看懂你的代码可不是件小事。这里边可能意味着要遵守公司的编码约定，或者要使用某个代码框架。但无论如何，专业的开发人员都必须熟练掌握类似的工具，因此这也算一条。

千万不要因为我没有列举某种工具或技术，就认为它不包含在内。要是你有疑问，可以这样想：它有没有提高我的工作效率，降低我的风险，或者增进我与他人的理解？如果答案是肯定的，那这种工具和技术就是我们所说的业内发展成果。

2.2 怀疑者指的是谁

所谓怀疑者，基本上就是你的同事，他们没有使用你希望他们使用的工具或技术。至于为什么不用，有些人属于不知情，有些人属于不在乎，而有些人则是知道但拒绝使用。怀疑者是有模式之分的，后面我们将会详细讨论各种怀疑模式。



关于这些怀疑者，最重要的是要搞清楚为什么他们还没有使用某种技术，以及为什么他们会对我们的好意不屑一顾。原因很多，有技术上的，有出于小团体利益考虑的，甚至还有个人好恶的因素。（有点匪夷所思吧？）为此，最关键的是要站在他们的立场上，找到他们之所以怀疑的思想根源。

在这里，我特意挑选了“怀疑者”这么个词，而不是感情色彩更浓的不识好歹的人、鼠目寸光的人、心怀敌意的人，甚至你能想到的更激烈的词汇。在身心俱疲的时候，我们很容易把他们当成赛场上的对手、反对派，甚至是敌对分子。偶尔这么想一想倒也能让自己发泄发泄，但千万不能陷入这种思维定式。他们都是你的同事、你的朋友，也许你过去还对他们中的某个人扮演过类似的角色呢。这一点可是不能忽视的。

2.3 为什么需要推销

推销这个词通常是指让人们为了得到某件东西而付钱给你。而说到推销业内发展成果，大多数情况下意味着要让人们付出另外一种代价。这种代价

就是时间和精力，即让人们投入时间和精力去学习新东西。而投入时间和精力到底能得到什么回报，有时候人们并不理解，尤其在他们正被手头的工具折腾得疲于奔命而又徒劳无功的情况下，就更不容易理解了。在疲于奔命的状态下，他们很难后退一步、更全面地去看问题：他们是在浪费时间，原因就是使用的方法和工具太不给力。

除了时间和精力，有时候还需要一些不太好说清楚的投入。程序员是喜欢拿语言来给自己贴标签的："我是Java程序员。"或者说："我是一个.NET程序员。"要让一个Java程序员使用Ruby，可远远不是光让他们花时间那么简单，还涉及他们对自己的重新定位问题，哪怕只是调整一点点。如果是让他们切换OS平台，那麻烦恐怕就更大得不得了了。

还有可能是其他更加说不清道不明的东西。当人们认为自己在某个领域已经可以"一览众山小"的时候，这些东西就会出现。比如说，不用再看参考文档的时候，或者在某个领域里已经摸爬滚打了10年之久了，也可能是拿到一个更高的学位之后。总之，会有一个标志性的事件作为转折点，过了这个转折点，有些人就会觉得自己无所不知了。你说的明明是："某某技术也许能改进他们的方法。"他们听到的却是："我认为你错了。"如果他们现在错了，大概过去这几年他们就一直那么错着。这顶大帽子，就算是最明白、最开通的人，恐怕也是轻易不能接受的，因为你不仅在混淆他们的身份，同时也在伤害他们的自尊。



上述种种迹象表明，你并不仅仅是想让他们掏点钱就完事了。你想让他们付出时间、精力、身份转换，乃至牺牲尊严的代价。这些东西哪一样都比钱更有价值。如果说为了得到钱必须得推销的话，那么为了得到这些东西，你推销的难度就会更大。

好了，现在你对工具和技术相关的问题已经有了明确的认识了。你知道了业内发展成果的含义，知道了自己工作的对象，也知道了为什么需要推销。但是，还有一个问题也不能不考虑：能不能一上来就推销你的工具和技术呢？下一章我们帮你分析这个问题。

第 3 章

解决正确的问题

在采取行动说服别人接受我们的方案之前，必须问自己一个非常重要的问题：我们是在解决问题，还是在推行方案？如果是解决问题，那很好；因为对于团队来说，你是在治病救人。如果是推行方案，那最多只是一种中立的情况，通常不会有人欢迎。然而，实际上推行方案的情况却屡见不鲜。这又是为什么呢？



当你发现了解决方案，为之兴奋莫名之时，往往就会忽略一点，即你的目的是要解决问题。而且你还忘了大多数问题都不止有一个解决方案。结果，你专心致志地推行自己的解决方案，而这个解决方案可能并不适合要解决的特定问题。尽管你推荐的工具确实也是解决问题的一个途径，但从团队的技术氛围、能力组合以及组织策略这几方面综合来看，很可能还有更好的方案可供选择。

就像我们说服别人时不希望人家固执己见一样，我们自己也要持开放的态度。为此，就必须保证自己的解决方案真的合适，再也找不到其他更好的方案了。只有这样你才有可能放弃自己偏爱的方案，信心十足地去推行最适合团队的方案。

Rails 开路

Chris是Java开发人员，但突然间红遍社区的Ruby让他移情别恋了。Ruby on Rails[①]那个“10分钟构建博客”的演示，让他着了魔似地迷上

① Rails是一个使用Ruby语言的快速Web应用开发框架。要了解更多信息，请参考*Agile Web Development with Rails, 3rd Edition* [RTH08]。

了Ruby。他爱死Ruby了。Rails真的让他工作效率倍增。他迫不急待地想让自己使用Java的同事们都赶紧试一试。

他的同事们也确实需要Rails。大家一直在使用公司自己开发的框架逐个逐个地写应用，但使用那个框架必须做很多琐碎的工作。结果，很多时间都花在了编写重复的代码上，而花在设计界面或可持续的模型上面的时间却少得可怜。

他刚一跟大家提出这件事，就有很多人抵制。没有人想学习一门新语言，或者一个新框架。有人已经向他们灌输了一些关于Ruby的FUD[①]，他们认为使用基架（scaffolding）固然好，但只是为了使用它必须新学一门语言，这代价有点太高了。

Chris问了一圈，发现团队中不少人或多或少都关注过Groovy。Chris知道有个Grails项目，他又花了一些时间熟悉Grails。有Java和Rails经验，学习Grails并不难。



他又劝说大家来了，但这次推销的是Grails[②]。以前因为Ruby FUD而心存畏惧的怀疑派不见了，语言的问题已经不存在了。唯一的问题就是要学习一个新框架。对于这个问题，团队成员认为基架能提高工作效率，还是值得尝试的。

他们接下来的项目就是使用Grails开发的，现在，这个团队已经离不开Grails了。

这个例子表明，Chris曾经想要兜售Ruby on Rails。而他们团队面临的问题是需要找到一种方法，既能提高工作效率降低劳动强度，又可以把精力主要集中到真正创造价值的地方。Ruby on Rails只是解决这个问题的众多方案之一。重写公司自己开发的框架，增加代码生成功能，或者找一个能简化数据建模的IDE，乃至使用Grails，这些都是可能的解决方案。为此关键是要迈出下一步——寻找其他方案，然后客观地权衡利弊。

---

① FUD，即Fear（害怕）、Uncertainty（不确定性）和Doubt（怀疑）；详见第17章的解释。

② Grails是一个类似于Rails的快速Web应用框架，但使用的是Groovy语言。要了解更多信息，请参考*Getting Started with Grails* [Rud07]。

3.1 为什么要布道

之所以说要在推销方案前先努力想清楚问题所在，主要的原因有以下几个：

- 这样做可以先让自己弄明白到底是不是真的存在问题；
- 这样做可以强制你站在听众的角度来思考问题；
- 这样做可以让你拿出最适合听众的解决方案。

甚至，你都得问问自己，到底有没有需要解决的问题？在前面那个小故事里，确实还真有一个问题。也可能那个项目团队已经在用Grails了。如果遇到这种情况，你就必须扪心自问，Ruby on Rails能给那个团队带来些什么？说实在的，技术平台移迁就算花钱也花不了太多。此时此刻，最要紧是得搞明白，你是真的想帮团队成员提升工作效率呢，还是只是希望说服别人跟你一样成为某种新技术的粉丝。只有确定真的有问题，才能弄清楚是什么问题，以及这个问题是否值得解决。



确定了真的存在问题之后，接下来就要考虑它是否值得解决，或者说值得你的团队兴师动众的到底是什么。也许团队要做的事太多了，也许他们只不过要将这些难题都推给那些新成员，想让新成员在实践中得到锻炼；那些老资格的人则专注于模型和UI。在这种情况下，这个团队的解决方案反倒比你所预测的更有价值。你必须给自己推销方案换个更合适的理由，同时还必须考虑到，假如把难题解决了，那些失去锻炼机会的新成员又该怎么办。

此外，在问题存在一个现成解决方案的情形下，还必须清楚你要推行的是不是一个定制的方案。无论是干裁缝，还是在IT行业，定制的方案总是能要到更高的价钱。原因其实都一样，定制的方案是“量体裁衣”，本身就已经埋下了成功的种子。为团队专门量身打造一套合适的方案，可以减少推行过程中的摩擦，或者一些磕磕绊绊。

3.2 考虑方案

要解决正确的问题，最困难的莫过于把目光从自己最倾心的方案转移到其他方案了。为此，必须要有兼容并包的胸怀，勇于放弃先入为主的成见。做到以下几点，会更容易克服这个困难。

3.2.1　研究问题

对于要解决的问题，参考一下其他组织的解决方案。在寻找解决方案时，记住要从问题出发，而不要从特定的实现方法出发。

具体来说，要寻找：

- 源代码控制工具，而不是SVN；
- 快速应用程序开发手段，而不是Ruby on Rails；
- 富Internet应用开发框架，而不是Flex；
- 对象关系映射库，而不是Hibernate。

但有时候要做到这一点是不可能的。问题很明确，但就是找不到相应的解决方案。怎么办？可以考虑另一种技术方案，比如：

- OS X中类似Visual Studio的工具；
- Exchange的开源版本；
- Linux中的Safari。



3.2.2　摸底调查

接下来要做的就是对团队的技术能力和思想倾向做一次摸底调查。像前面故事中的Chris那样，找几个人聊一聊。看看大家了解什么情况，对问题有什么看法。在此期间，可以验证你的一些猜疑，或者，找到一种完全不同的解决问题的途径。不管怎么样，即使摸底调查没有给你带来什么灵感，但至少可以通过谈话了解人们当前的困惑程度，知道他们可能接受你提出的什么解决方案。

3.2.3　开列清单

强迫自己列出各种可能的替代方案，就算不是太好也不要紧。总之，你得把自己调研过的那些方案一一地列出来，以便自己参考。如果你跟别人说：“没有别的替代方案。”人家不会相信你。你可以说自己的方案无与伦比，但却不太可能说它是唯一的解决之道。替代方案始终都会有的，即使你到头来不会采用，但至少还得考虑它们。

况且，列出替代方案反而有助于证明你的观点更有吸引力。开列清单本身就说明你研究过了，也精心准备过了，绝非随随便便地碰上一个算一个。总之，能够让你的听众们感觉眼前一亮。

3.3 面临的挑战

这时候的主要挑战就是做不到真正客观地考虑替代方案。越是公平公正地对待替代方案，就越能像第14章概括的那样充满战斗激情，你的心胸也就会更加宽广，掉进这个陷阱的可能性就越低。

另一个困难是这个过程需要耗费时间。如果公司需要你尽快拿出方案来，那恐怕你来不及充分调研。不过，这种风险比较低，因为只要你能说明白尚未确定哪个方案更好，大多数组织都会在全面实施新方案以前选择维持现状。

3.4 几点建议



假如你在评估替代方案的时候遇到了麻烦，可以试一试下面几条建议。

- ❑ 在拿出你的解决方案之前，先做一番研究，看看那些创建方案的人是想用它解决什么问题。
- ❑ 学习一种能替代你的解决方案的技术。不一定非要成为该技术的专家，但至少要能搞明白替代方案是怎么回事。
- ❑ 扮演怀疑者的角色。想象一下你很反感这个解决方案，因此想要竭尽所能来阻止公司实施这个方案。你反对的理由是什么？然后再想一想，现实当中是不是真有这种合乎常理的因素。

显然，你应该让自己推动的解决方案能够适合公司并真正对公司有帮助，但实际上很多人都做不到这一点，并且还遭受了不应有的苦难。在我们这个行业里，由于决策者自己陷入困境而没有找准要解决的问题，导致项目成员度日如年的例子已经太多了。亲爱的读者，你千万不要做这种人，一定要推行那些真正对组织有用的技术和工具。这样不仅会让自己更容易达成目标，更重要的是，你在做正确的事。

Part 2

第二部分

怀疑者模式

本部分内容

第4章 了解身边的人

接下来的几章我们介绍几个模式，也就是那些质疑技术变革的人可以归入的类别。通过了解这些模式，你就可以分辨出自己身边的人有谁属于这些模式。

知道了要跟谁打交道，才能找出与其打交道的方法。后面，我们还会针对这些怀疑者模式给出一系列应对的技巧。不过首先，必须得知道他们是谁。

我们要介绍的怀疑者模式包括：

- 孤陋寡闻型（uninformed）；
- 随波逐流型（herd）；
- 百般挑剔型（cynic）；
- 激情燃尽型（burned）；
- 时间紧迫型（time crunched）；
- 发号施令型（boss）；
- 不可理喻型（irrational）。

一开始，要弄清楚谁属于哪种模式恐怕有点困难。看着每种模式的说明，你心里可能会嘀咕："George有点喜欢挑刺儿，可他还没有那么离谱。"这说明George属于百般挑剔型，但还不严重。本书中描绘的这些模式，多少都有点演绎，因为我们只是想说明问题。这里面最关键的是要看有没有那种行为，而不是看行为的严重程度。在现实当中，相信大多数人都会比较专业，不会像我们这里给出的有些夸张的例子中所描述的那样。



同样，也不要为了确定某个人到底属于百般挑剔型还是激情燃尽型而死盯着细节不放。每个人都有可能同时属于不止一个怀疑者模式，这很正常。比如，发号施令型和时间紧迫型的特点经常会在同一个人身上体现出来。对一种技术漠不关心而百般挑剔的人们，对另一种技术而言，往往又是激情燃尽型的。这是一件好事，因为这样一来，大多数怀疑者模式都会有一些通用的制胜技巧。对这些多重类型的怀疑者，可以集中应用通用技巧，而不必费额外的工夫。

假如你觉得这些类型的人实在不好区分，可以参考如下提示：

- 孤陋寡闻型的人不可能同时是激情燃尽型的人；
- 大多数人多少都有一点时间紧迫症；
- 很难把随波逐流的人定性为怀疑者；
- 不可理喻型的人经常会伪装成其他怀疑者。

好吧，现在就来看看我们身边这些每天都会见面的、普普通通的怀疑者。



第 5 章

孤陋寡闻型

周五晚上六点钟，你在帮一位同事的忙，他的硬盘昨天坏了。他已经重装了机器，也恢复了备份，可他周三、周四干的那些活儿都丢了——其中还包括修复一些Bug的重要代码。你提醒他，这些代码是可以从源代码控制服务器上取回来的。他反问道：

“什么是源代码控制？”

可想而知，他没有使用源代码控制；事实上他根本就没有听说过源代码控制。他见过你发的那些关于新Subversion[①]服务器的邮件，可见过并不等于就看过啊。他们并不认为Subversion服务器是自己应该关心的事儿。

那些口口声声“无知是福”的人，从来不知道在周五晚上七点，绞尽脑汁把从备份中恢复的内容与一个名叫“main – copy20070811”的文件夹对应起来是一种什么滋味。而我们呢，我们知道无知并不是福。我们要聆听自己同事的抱怨，因为“办公专家应该知道怎么不让他们犯那么愚蠢的错误”，然后忙到晚上九点回家。

5.1 他们为什么不用这种技术

原因通常是他们不了解这种技术。有些人是从来就没有听说过，而有些人可能知道有这种技术，但不清楚它能解决什么问题。更重要的是，他们并没有意识到自己就有要用这种技术解决的问题。

不管是什么原因，总之就是他们有需求，但自己却浑然不觉。

① Subversion是一个集中式的版本控制系统。要了解更多信息，请参考*Pragmatic Version Control using Subversion, 2nd Edition* [Mas06]。



5.2 深层次原因

人们不了解某种技术的原因有很多。并不是所有人都会看博客，跟踪行业最佳实践，由此造成的无知有时候是有意为之的结果。某些人是地地道道的“朝九晚五”式上班族，他们不在乎提高自己的技能。不过，更多时候恐怕还要归咎于我们这个行业。这个行业中的信息多得令人目不暇接，除非你有目的地去找，否则不可能什么都知道。

5.3 有效的应对策略

孤陋寡闻型的人一般不会“自己走向山”，因此你必须“把山唤到他面前”。[①]对付孤陋寡闻型的人，最有效的策略就是给他们提供信息，以下几章都有介绍：



- 第13章，“取得经验”；
- 第14章，“传达理念”；
- 第3章，“解决正确的问题”；
- 第15章，“展示技术”；
- 第18章，“公之于众”。

5.4 几句忠告

首先说好消息：把孤陋寡闻型的人转变为其他怀疑者模式的人极其容易。所要做的，无非就是告诉他们这种技术——唰地一下，他们就不再孤陋寡闻了。说说坏消息吧：他们很可能会皈依别的怀疑者模式，而不是被你说服。

---

① 这句话源自穆罕默德唤山的故事。先知穆罕默德，带着他的四十门徒在山谷里讲道，他说，“信心”是成就任何事物的关键。一位门徒对他说：“你有信心，你能让那座山过来，让我们站在山顶吗？”穆罕默德对他的门徒满怀信心地把头一点，对山大喊一声：“山，你过来！”山谷里响起了他的回声，回声终于消失，山谷又归宁静。大家都聚精会神地望着那座山，穆罕默德说：“山不过来，我们过去吧！”他们开始爬山，经过一番努力，到了山顶，他们因信心使希望实现而欢呼。这是人们熟知的一个典故。山并没有过来，穆罕默德却翻过了那道山。这个典故提示人们，在改变不了客观环境的时候，人要改变自己的主观信念。唤山不来，可以走向山。——译者注（摘自网络）

为此，我不会指望跟他们坐下来，聊聊什么是版本控制或单元测试就让他们心服口服。除此之外，必须要使用反击手段，尤其是“展示技术”。不仅要向这些人解释清楚这是一种什么样的技术，更要摆出采用这种技术的理由。



第 6 章

随波逐流型

今天是代码评审的日子。花费大量时间翻阅了整整200页打印出来的代码之后，你发现其中大部分都是常规性的CRUD[①]代码，完全可以拿一些通用对象或某种ORM[②]方案取而代之。你可以预见，在代码评审中提出重大改进意见将带来激烈的争吵。

这是Hector与团队合作的第一个重要应用。他刚刚走出校门不久，而这是他为公司开发出的第一个重要成果。评审会一开始，Hector就有点如坐针毡，而你担心的是在告诉他“用几个通用对象就可以减少75%的代码”之后，他会有什么反应。

这一刻到来了，你就在他对面。然而，没有生气，没有争吵，也没有怒发冲冠掀翻桌子，他只是那么直勾勾地看着你，说：“我可以那样做吗？当时我也想过，可我不知道你们会同意我那么做。”他的回答令人震撼，你怀疑自己是不是听错了：居然有人明知道怎么做才是对的，但还是按照自己觉得不好的方式去写代码，浪费时间！这种现象在普通开发人员中是相当普遍的。



6.1 深层次原因

随波逐流型的人之所以随波逐流，因为他们的身份是追随者，而不是领导者。对于一个组织来说，保持领导者与追随者之间的平衡至关重要。领导

① 表示Create、Read、Update和Delete（创建、读取、更新和删除）。这四种操作在任何使用数据库的应用中都是最基本的。

② 即Object Relational Mapping（对象关系映射），是一种在面向对象系统中将数据记录映射为数据库对象的技术。

太多，组织的发展反而会举步维艰。为什么？每个领导都希望团队去实现自己的思路。不过在实际当中，还是追随者太多而领导者太少的情况更为常见。在这种情况下，前面故事中提到的情景就会出现。说来也怪，随波逐流型的人虽然通常是追随者，但他们有时候也会成为团队的领导，结果当然无所建树了。

随波逐流型的人，也分为两种：

- 不知道自己有领导能力的人；
- 不以当领导为目标的人。

不知道自己可以当领导的人一般来讲都比较年轻。他们走上工作岗位的时间不长，缺乏自我激励的人生经验。这些人很有想法，也愿意去尝试新鲜的事物，可他们尚未领会到做领导的要诀：只要有人提拔，你就可以做领导，但能否成为领袖，却不是由别人说了算的。



不以当领导为目标的人经常会受到激进程序员的鄙视。这些人朝九晚五，循规蹈矩，靠工作维持生活，只在上班时间工作，下班回家就一心扑在自己的其他追求上：家庭、爱好、社区，等等。他们不会主动意识到一直学习最新技术有什么好处。

6.2 有效的应对策略

随波逐流型的人不会主动寻求领导的帮助。你必须主动找他们，向他们提要求。激励他们进步的手段会有效果，但十分有限，而某些强制性措施或许对他们更好使。而且，随波逐流型的人不会对强制他们进步有抵触情绪，这一点与其他类型的人有所不同。他们愿意被人领导。因此对这种类型的人有效的策略很多。以下各章都给出了针对随波逐流型的人的一些有效策略：

- 第13章，“取得经验”；
- 第15章，“展示技术”；
- 第21章，“来点刺激”；
- 第20章，“搭一座桥”；
- 第18章，“公之于众”。

6.3　几句忠告

随波逐流型的人是第二容易被说服的类型。基本上就是你带领他们，他们跟在你后面。虽然简单但仍然要付出努力，你必须得运用自己的领导力。看起来这点代价算不了什么，也很直截了当，但领导工作会使人烦躁不安。举个例子，可以把这种工作当成维护应用程序。相对于构建应用程序来说，维护应用程序的代价总是要高得多。不间断的领导工作或做思想工作的过程，需要付出比刚开始说服他们多得多的努力。我并不是想打消你的积极性，只想提醒你做好这件事也不容易。当然，要是你能发现并简拔几位年轻人加入你的队伍，这份投资还是相当划算的。



第 7 章

百般挑剔型

你刚刚给自己的团队作了一场20分钟的展示，关于源代码控制。是，他们根本就没有做源代码控制。没错，就是一丁点儿都没有，似乎近20年来计算机科学就从未有过任何发展成果。不过，晚采用总比不采用好，更何况只要做源代码控制，就比不做好，因为一次误删就可能酿成灾难。你已经做了充足的准备，而且自己也反复测试过了。一个方案不理想，接着再试验下一个方案。试验来试验去，考虑到当前这个特定的用户群、方案的成熟度、工具的易于获取，以及行业的认可度，最终你还是觉得Subversion合适。不管怎样，你的方案已经讲完了，现在该回答别人的问题了。



“我听说Subversion已经不怎么流行了，随着Git正逐渐成为源代码控制领域的下一个焦点，难道我们选择Subversion就是为了先撑两年吗？”Cindy问。

对此你已经胸有成竹：“问得好，Cindy。我比较了很多方案，也包括Git。最关键的是，我们这个小组需要能够集成到Eclipse的工具。我个人认为针对Eclipse的Git工具还没有出现。一两年以后也许会有，但我们对解决方案的需求是迫在眉睫的。因此，目前对我们而言，Subversion就是一个合适的解决方案。假如我们真的需要Git那样的功能，也有很多工具可以从Subversion迁移到Git，所以我并不担心这个问题。”

你想当然地认为，如此滔滔不绝地讲完之后，这个问题就算解决了。可就在这时候，Cindy又抛出一个问题来。

“我听说Subversion会向项目里添加各种元数据，项目越大，被加入的元

数据就越多。而且，我听说这种元数据很容易因为没人管而变成垃圾。”

“确实存在这个问题，不过好在我们的项目都不太大，所以也不会有什么严重问题。对于我们那些比较大的项目，很多前人总结的最佳实践都可以参照和借鉴。”你杀了个回马枪。

接着又这么来来回回地过了几招儿。有时候，你能把问题解释清楚，但有时候也说不清楚。有时候，你甚至会想Cindy是不是在维基百科中找到Subversion条目，然后只看了“目前的问题和局限性”一节的内容。要不然，一个从来没有接触过Subversion的人，怎么会知道它的那么多缺点呢？

最后，Cindy用貌似权威的语调宣布：“既然问题这么多，而且我们一直没用它也都过来了，所以我认为没有理由给我们的开发环境增加额外的复杂性。”



在场的其他人也许不完全同意Cindy的这个结论，可他们坐在那看你们俩人打口水仗已经好一会儿了，大家的咖啡可没少喝。你跟Cindy唇枪舌箭，短兵相接，考验了大家的耐心，也考验了他们的膀胱。原本该是一次精彩的技术布道秀，现在反倒闹成了人事斗争。

这样的场面，只要面对百般挑剔的人，就会一次一次上演。

7.1 深层次原因

这种行为的原因有很多。有些人可能就是喜欢争论。还有些人喜欢证明自己比别人聪明。再有一些人可能已经在行业里混迹了一些年头了，但一直四处碰壁，因而对任何事儿都不会看到其有利的一面。

但是，之所以你会碰到这样一种人，其实还有一个非常重要的原因，那就是：*在我们这个行业里，这种行为会有所收获*。

我们这个行业之所以能赚到钱，大多数都是建立在人们的想法之上的。聪明固然重要，但比*真正聪明*更重要的，则是*让人觉得聪明*。

真正聪明在哪里都可以一显身手，不一定非要当着观众的面去表演，所以也就经常不会被人注意到。实际上，置身众目睽睽之下，思想更容易不集

中，反而会影响人的发挥。每次跟某个小组讨论问题之前，你通常都要精心准备一番，正是这个原因。坐在自己私密的小隔间里，你完全可以相信自己到时候能够调动起每一个脑细胞来。可一旦你面对一群人，各种预料不到的情况就会出现，加上自己精神紧张，有时候再碰上坏运气，结果就会导致你无法发挥出自己的全部潜能。

苛求是好的，有意挑刺就不好了

咬文嚼字可不是件容易的事。对某一个人的有意挑刺，对另一个人而言或许就是尽职调查。我不希望你把百般挑剔的人提出的每一条批评意见都看成是一次攻击。为你选择的工具或技术辩护是你的责任。谁都不应该盲目地接受别人替自己选择的哪怕是正确的道路。

然而，我在这里把批评说成是挑刺并非为那些不称职的布道者开脱。挑刺的意思是指纯粹为了阻挠而阻挠，或者说就是想要给人出难题。这样说应该可以帮你辨别这种类型的人了。实在不行，你还可以参考 Potter Stewart 在 1964 年说过的关于猥亵的一句话：“是不是猥亵，我一看便知。”*百般挑剔也是如此，它很难定义，但很容易感觉到；有时候，一些人提出反对意见是为了给所有人争取一个最好的结果，而有时候一些人百般诘难，其实是想逃避自己的进步，或者是以其他人为代价来伪装自己的聪明。

* Potter Stewart 当时是美国最高法院大法官，在判定一起有关猥亵的案件时，曾说过一段著名的话：“今天，我不会再尝试按照我认为能够理解的方式对这个概念（‘赤裸裸的色情描写’）下定义；也许我永远也找不到一个能让人容易理解的说法。不过，是不是猥亵，我一看便知。”



而另一方面，看起来聪明，则只需要有观众并能给人一种聪明的印象，就不必费力劳神地实际去做出来了。

这跟为数众多的百般挑剔型的人有什么关系呢？有两种方式可以让人看起来聪明：

- 在观众面前表现得非常聪明（这一点我们已经说过了，很难做到）；
- 在观众面前表现得比某些人聪明。

第二条比较容易做到，而这也正是百般挑剔型的人最常用的方式。你提出的每一个观点，他都要反驳一下，给你施加压力，以示自己并不比你考虑得少，这样他们看起来就很聪明了。只要从你准备的材料中找出一个弱点来，他们就可以借题发挥，让自己看起来比你厉害。

7.2 有效的应对策略

应对百般挑剔型的人，无非两方面：

- 让他们少开口，别给他们机会；
- 作好充分准备，避免被其驳倒。

让他们少张口，首先就是不能让他们逮着任何提问题的机会。比如，你可以在说话时让自己的语气保持连贯，可以提前设想一些可能的问题，然后在宣讲的过程中自问自答。如果你都已经做好准备，而他们真的提出这个问题，那就先请他们稍安勿躁，告诉他们答案已经在你的资料中。只有这样，你才能控制整个对话的局面，而不致喧宾夺主。



从另一方面讲，准备得再充分也不可能万无一失。谁也不可能想到所有的问题。首先，要用不容置疑的口气来回答他们的问题；就算你不知道怎么回答，也必须非常自信地说出“我不知道”这四个字。然后，要尽量全面了解相关主题的知识，用这些知识来武装自己，不让他们通过出你的洋相而抬高自己的阴谋得逞。

为了能顺利地把以上策略付诸实践，以下这些对付怀疑者的方法也都十分有效：

- 第13章，“取得经验”；
- 第14章，“传达理念”；
- 第 3 章，“解决正确的问题”；
- 第15章，“展示技术”；
- 第21章，“来点刺激”；
- 第18章，“公之于众”；
- 第20章，“搭一座桥”。

7.3 几句忠告

假设你像前面所说的那样，准备充分而且驳倒了他们，那你通常也会赢得整个组的尊重。通过你的传达方式和专业内容，你的方案也会成为聪明的选择。既然选择这个方案很聪明，那么质疑它的人就会显得不够聪明。在不愿被人看扁的心理压力下，他们自然也就不会再找你的麻烦了。没准，要是你能把自己的方案包装得足够好，他们还能成为你最有力的支持者。



第 8 章

激情燃尽型

几位开发人员已经凑成了一个小组，开始制定下一个项目的计划了。此时此刻，正好是说服他们采用某些工具和技术的最佳时机。在公司的某个项目中推广Hibernate可是你一直以来的梦想，而眼下这个项目再合适不过了。这个项目要为一个简单的数据库开发一个巨大的前端管理系统。它足够复杂，完全有使用ORM的必要；同时，它又不是特别复杂，项目团队不会因为它而面临一些想象不到的难题。



你提出了使用Hibernate[①]的建议。

“我以前用过Hibernate……”Bernard响应道。

你简直欣喜若狂；说服这个组已经问题不大了。只要Bernard站在你这一边，那就不用费事了。

Bernard继续说：“太可怕了。整个系统一下子就不动了。除了添乱一点用都没有。要做一次更新，都得从数据库中查出一整条记录，还要外带其他表中有关系的行。这么多工夫实际上却只完成了一条简单的更新语句的工作。所以，对你的好意，我只能说不！”

就像Lucy对Charlie Brown那样，Bernard刚为你点燃了希望之火，随即一口气又把它吹灭了。他的负面评价基本上已经不可能让你对这个组的说服工作再向前迈进一步了——不是因为他说的一定对，而是因为他有经验。他曾经被这种技术伤害过，如今他不想再重温旧梦了。

① 针对Java的一个便利的ORM（Object Relational Mapping，对象关系映射）框架。

厌倦变化型

在刚开始写这本书的时候，我曾考虑过另外一个怀疑者类型，叫厌倦变化型。想来想去，最终还是觉得“厌倦变化”其实就是激情燃尽型。

好的变化会让人付出很多，但通常都不至于令人消沉。变化之后人们想的只是休息一下，而不是永远地休息下去。坏的变化会导致人厌倦变化，但更常见的是，坏的变化只会造就让人一眼就能看出来的激情燃尽型的人。

激情燃尽型的根本问题在于，他们厌倦变化并不是因为变化的好或坏，而是因为变化除了变化本身之外，什么都不是。于是，他们的激情，没有了往日的温度。假使他们以前没有经历过类似的事情，这会令他们心生厌烦；如果他们已经经历过了类似的事情，那要说服他们将势比登天。

不会带来积极结果的变化就是坏的变化，哪怕不是什么灾难，而仅仅只把人搞得哈欠连连。所以，厌倦变化就是激情燃尽，即便是激情慢慢燃尽。



8.1 深层次原因

这种类型的人很容易识别。导致他们没有激情的原因，是他们曾经用过你打算推行的工具，但那工具不好使。他们在使用工具时遇到了什么问题？什么问题都有可能。可能是使用了该技术，但进步不大。也可能是因为使用它而导致了严重的挫败。抑或只是安装过，但之后不知道怎么使用它而已。不管是哪种可能的情况，也无论他们尝试的深度，对其他人而言，他们的话都是可信的，因为他们拥有第一手的经验。结果，不仅他们怀疑的态度更容易被人接受，而且也为你反驳平添了不少难度。

8.2 有效的应对策略

要想应对激情燃尽型的人，最关键的是要弄明白他们经历过什么事。要是有使用Hibernate的经验，那你就会知道Bernard曾经遇到的问题了。也许，在之前使用Hibernate的时候，有人可能在取得数据时搞错了方式，或者延迟

加载弄得不知道哪儿不合适了。这个人可能就是Bernard，也可能是别人，但最终的结果没有什么不同——他们对Hibernate的实现存在严重的缺陷。这个例子本身可能不足以说明一切，但也很有可能Hibernate对之前的项目根本就不合适。试想，选择了不合适的工具，即便配置使用方法都正确，Hibernate也逃脱不了被摒弃的命运。

因此其次，你必须得知道自己建议的方案是不是用得其所。这个工具真的值得你去扭转激情燃尽型的人的成见吗？是否存在一个更好的技术，能让激情燃尽的人更容易接受它，同时还能实现类似的功能？

最后，你要尽力把自己的所思所想讲给激情燃尽型的人听。你得想方设法让他们明白自己从前的实现是有问题的，并且不要涉及会让他们神经紧张的话题。他们必须信任你才行。在与他们沟通交流的过程中，必须小心翼翼地平衡正反两方面的说法。



在这些条件下，可以参考下列技巧：

- 第 3 章，“解决正确的问题”；
- 第13章，“取得经验”；
- 第14章，“传达理念”；
- 第15章，“展示技术”；
- 第21章，“来点刺激”；
- 第18章，“公之于众”；
- 第17章，“建立信任”；
- 第20章，“搭一座桥”。

8.3 几句忠告

要想让这种人完全转变观念是相当困难的。人本来就不太喜欢花时间学习新东西。曾经尝试过的新玩艺，让自己吃到了苦果，而今要再去尝试一遍，怎能不进行激烈的思想斗争呢？有时候选择一项替代性的技术，倒是比较容易驱散人们“重温噩梦”的阴影。不过，经过不懈努力，加上其他人的支持，对于正常的激情燃尽型的人，还是能让他们回心转意的。

第 9 章

时间紧迫型

我们继续以第1章中提到的那次会议为例，在那次会上，我在跟其他经理讨论是否需要把应用服务器升级到新版本。

反对的主要原因是他们对代码已经有了自信。如果迁移到另一个服务器上，就得重新测试，重新测试就有可能漏掉Bug，而漏掉Bug又会让他们在客户面前丢脸。我对这个问题给出的解决方案是单元测试，但听到的第一个拒绝的理由就是“单元测试太费时间”。



事实上，我跟这个团队之间的每一次争论，最终都会跑到时间不够用这个老生常谈的话题上来。就算单元测试能缓解该团队面临的许多其他问题，就算单元测试可以很大程度上解决导致时间不够的那些麻烦，他们都无动于衷。这些人思想上存在的问题就是：“没时间去做正确的事，但有时间重复做错误的事。”

9.1 深层次原因

说时间不够用的，往往是那些目光短浅、“丢了西瓜捡芝麻”的人，或者那些看不清问题实际成本的人。平心而论，让谁站在那些人的立场上，可能也都很难看清问题所在。但人们多数时候都不愿意改变计划来适应新事物，尤其是在他们对结果心存疑虑的情况下。这样说吧，他们现在的方法可能会浪费时间，但他们清楚地知道会浪费多少时间。而新方法肯定是要花时间去学习的，但付出学习的时间就一定能得到相应的回报吗？不一定。反而有可能会因此拖他们的后腿，导致时间比以前更加紧张。

进一步看，有些人之所以感到时间紧迫，是因为他们的工作量没有足够的资源保证。缺少人手，过度承诺，以及其他形式的计划不周，都是造成这种局面的原因。这些人无时无刻不感到时间紧迫，要想让他们挤出点时间来，真要比登天还难。

另外一些人呢，并不是时刻都感觉到时间紧迫。很多行业的需求都有周期性的特点，他们也会因此忙一阵，闲一阵。就拿美国的高等教育行业来说吧，主要的学期从每年9月份开始，12月份结束。然后会放一小段假，随后就是1月到5月的春季学期。接下来又放一小段假，然后是较短的夏季学期。在学校里开发软件的人，上学期的9月份都是忙得脚不沾地儿的。在冬季假期中，他们也忙于升级。之后，他们会在夏季投入到新项目中。一年中的其他时间，他们只是维护自己的软件或者给软件打一些补丁。向这些人推销新的想法，最合适的时机是他们比较悠闲的时候，而不是他们忙得焦头烂额的时候。换句话说，8月份的时候，向这些时间紧迫型的人推销，一般都能得到他们的积极回应。总之，造成这种局面的原因是他们的工作具有周期性，并非他们的思想有什么问题。



最后，的的确确是有那么一些工作属于对时间要求比较苛刻的类型。按期完成这些工作是最重要的，至于完成工作的方法是不是最合适，就较次要了。有一次，我为本地负责竞选的一个办事处开发软件。他们需要一个最基本的CRUD应用，当时距离“选举日”只有24天。而“选举日”一过，那个应用就没什么用了。对这项工作而言，考虑什么可维护性或者可扩展性，都是多此一举。而要向负责类似工作的人推销这种功能，也只能是枉费心机，徒劳无功。

因此，问题的关键在于搞清楚时间紧迫到底是什么原因造成的：人的思想有问题？你选择的时机不合适？还是那个项目的类型不合适？第一种情况居多。但是，同样也要有这个意识——不要在不恰当的时机，或者向不恰当的人推销，这样会节省你的宝贵时间。

9.2　有效的应对策略

要想成功地说服时间紧迫型的人，就得让他们认识到你推销的技术或工

具确实能够减轻他们的负担。他们不光要听，还要眼见为实，有时候可能还需要利用其他节省时间的方式打动他们。无论如何，都得让他们相信，只要选择了你推荐的方法，一定能节省出比投入的多的时间。为此，以下几章提到的方法能派上用场：

- 第3章，“解决正确的问题”；
- 第15章，“展示技术”；
- 第16章，“适当妥协”；
- 第21章，“来点刺激”；
- 第20章，“搭一座桥”；
- 第19章，“注重合力”。

9.3 几句忠告

没有人想被时间逼着做事。这是一种非常不爽的感觉，时不时地会让人进退维谷。谁不想摆脱这种境地呢？他们可能只是不知道怎么做才好罢了。

不管怎么说，对付这些人，就是要讲条件的。你只要能给他们节省时间，他们就愿意跟你合作。最难的地方就是让他们相信一定能够节省时间。不过，要是真让他们相信了这一点，那他们急于走出时间困境的心理对你就十分有利了。



第 10 章

发号施令型

老板让你到她的办公室开个例会。会议一如往常，她讲，你听；你讲，她玩黑莓。你谈了当前正在负责的几个项目，还有那些已经完工的项目。把这些该说的都说了之后，你觉得该跟她谈谈自己那个小项目了。

你自己一直都在搞一个自动化引擎项目，它可以分析数据库，构建CRUD应用，并让这个应用具备公司要求的统一的外观。等创建好这个项目后，你打算花点时间调整一下它的模型，修饰一下它的UI，让这个应用更趋完美。虽然这个辅助应用很简单，但它真的非常棒。



你把这些也跟老板说了，她似乎没听明白，只问你在这上面花了多少工夫了。你如实回答，原来不怎么感兴趣的她，莫名其妙地恼怒起来。她说，你赶紧把这个事儿停下来，别继续做了。然后她就让你走，好跟其他下属开会。

你一下子从信心满满变得垂头丧气，怅然若失地走出老板办公室。以上情景通常会在你向管理者推荐一些专业技术的时候发生。

10.1 深层次原因

管理者抗拒专业技术的一个最大的原因，就是他们不理解这些技术。这话可能有点不好听，但实际上他们就是不理解。懂技术不属于他们的职责范围。即使是那些有程序员背景的管理者，只要他们不再做开发工作了，他们就可能会纳闷为什么你会需要那些他们从来用不上的东西。而如果他们在管理的同时还继续做开发，那他们几乎肯定是时间紧迫型的（参见第9章），所

以很有可能会对你推荐的技术置若罔闻。

这算不诚实吗

向管理者推荐技术，最难办的是找到与他们经历类似的场景，然后以他们听得进去的方式跟他们解释。可能有读者会拒绝接受我的这个建议，因为他们认为这听起来像是在骗人或者不诚实。可是，我们日常的行事方式确实是类似这样的。

即便是提供服务，收银员也会说“请”和“谢谢”。我们面试的时候一般会穿西装，而实际上大多数程序员都讨厌穿着西装写代码。如果你想到国外旅游，通常会先学几句那个国家的日常用语。

简言之，对别人投其所好，并用他们的语言跟他们沟通并不是不诚实，而是务实。在对别人有所求的情形下，必须要能够换位思考，而不是按照自己的意愿行事。至于管理者嘛，我们只不过是向比自己权力更大的人顺从一下而已。想一想这样做的价值吧，如果不这样做才是愚蠢的。



公平一点儿说，你自己本身也是问题的一部分。你也知道，开发人员有开发人员的问题。而你已经习惯了将自己推荐的工具作为解决开发人员问题的方案来讲解。管理人员呢，有他们的问题。因此，你必须把自己的工具作为解决管理者问题的方案来讲解。

在管理者面前，“可以让代码更容易维护”要说成“能够减少项目的成本”，而“自动生成机械代码”要说成“能够让项目更快结束”。没错，这样一来，就得说一些管理行话和市场营销用语，可是你也因此能达到自己的目的。

10.2 有效的应对策略

改变管理者的想法，归根结底还是改变你讲解自己工具和技术的方式。千万不要提代码行和N层数据库。而要说更少的人力投入和更少的停机时间。甚至，最好能用节省多少成本的数字来说话。

此外，还可以把你的解决方案同最让管理者棘手的问题挂起勾来。如果

必须遵守政府新出台的某项规章，而你的工具可以帮上忙，那你用这样的方式就一定可以说服管理者。最后，出于种种原因，管理者倾向于相信“外来的和尚会念经”，从而对内部的解决方案不够重视。如果能让“外来的和尚”替你的方案说几句话，那成功的可能性就会很大。考虑到上述这些情况，以下几章讨论的技巧在你对付管理人员时值得参考：

- 第14章，“传达理念”；
- 第15章，“展示技术”；
- 第19章，“注重合力”；
- 第18章，“公之于众”。

10.3　几句忠告

能否成功地说服管理人员，关键在于能否让他们把你推荐的工具当成解决他们问题的方案。一般来说，只要你能做到这一点，他们都会非常乐意接受你的建议。这可真是幸运的结果，毕竟管理人员可是有权力、最后拍板的人啊。他们可以强制让其他人接受你的方案。而这正是促使你的组织转变这个更大目标的重要步骤。



第 11 章

不可理喻型

不可理喻型的人是最难对付的一种类型。其他类型的怀疑者，至少他们的反对意见有一定的逻辑性和合理性。对于激情燃尽型的人而言，他们的前提是“这个以前失败过，所以这次也不会例外。”而对于时间紧迫型的人来说，他们的担心是“我现在有那么多的工作要做，哪还有时间来学习使用新技术啊。”这并不意味着他们是正确的，只不过是说他们的意见都有所依据。对于他们的假设和担心，你可以诘问、质疑，直到最终否定。你可以对激情燃尽型的人说，他们过去是失败过，但这并不代表使用你推荐的技术也会失败。你可以告诉时间紧迫型的人，他们完全可以拿出那点时间来，诸如此类。

但是，对于不可理喻型的人来说，这些都不管用。他们不想使用你推荐的技术或工具。原因可以说是五花八门，而且稍后我们会明白什么原因其实都无所谓，但就是没有合理的说法。跟他们争论不会解决任何问题。任何争论的目的都在于让你死了要说服他们的心。因此，这些人可能说出任何话来，总之就是想阻止你。

拜托了，Irene！

你又跟Irene争吵了一次。上一次你跟她争论的话题是数据库索引。她当时说索引是没有必要的，而且还说：“虽然损失了性能，但也避免了管理索引的麻烦，两者可以相抵。”你难以接受这种说法，现在还是接受不了。但这次争论的话题又改了。

今天争论的焦点在于你正在推动的新ORM系统。团队的其他成员都已经同意了，但Irene是项目经理，她拒绝了。



“我的团队成员没有时间学习新的系统；他们现在就已经忙得焦头烂额了。”

“在做上一个项目的时候，我在自己那部分用过它，省了我不少时间。”她的一个下属突然大声说，“如果整个项目组都用它，那就可以解决我们时间紧迫的问题了。”

Irene的眼睛眯了起来，她的说法已经不攻自破。她又想了一会儿，接着说：“ORM会带来性能损失。随着我们应用开发的不断推进，整体的性能损失是无法估量的。”

看了这个故事之后，你可能会想，她的理由和说法漏洞百出。你想的没错。可能你也注意到了，在一种说法被否定之后，她又会抛出一种新的说法。而且，这个新说法跟你和她之前争论时她的说法还相互抵触。这似乎暗示，她在每次争论时的出发点并不是她认为什么，而只是让你明白不要去烦她，好让她继续维持原状，不接受你的推荐。



11.1　深层次原因

导致人们不可理喻的潜在原因很多。可能是你和不可理喻型的人之间有什么过节，也可能是他内心更认同另一种技术或解决方案。不可理喻型的人还有可能是快跳槽了。抑或作为一名经理/开发人员，在两种意见争执不下的情况下，更情愿多一事不如少一事。

总的来说，具体是什么原因都不重要，因为行为都是一样的。不可理喻型的人会急切地跟你争论，伪装成其他怀疑者类型。如果他们的理由被驳倒，就换一个新理由；甚至也会变成另一种怀疑者角色。他们的内心都有无法明说的目的，而把这些真正的目的摆到台面上的可能性是极小的。如果这样的事真的发生了，那你就得准备打一场更艰苦的硬仗了。因为这个目的对他们来说太重要了，必须设法隐瞒起来，同时还要为了保护这个真实目的而百般掩饰，当然更少不了为维护该目的而极力争辩。

11.2　有效的应对策略

要对付不可理喻型的人，关键不在于转变他们的想法，而在于运用各种

工具和技巧将他们的反对意见公之于众，同时设法扭转他们的说法。但是不要搞错了，你不是要尽力说服他们，而是要尽力包容他们。以下两章包含如何对付不可理喻型人的一些技巧：

- ❑ 第14章，“传达理念”；
- ❑ 第17章，“建立信任”。

不要招惹狂热分子

有那么一类人，我喜欢称呼他们为“狂热分子”。狂热分子也属于不可理喻型，因为你不可能通过理性的分析说服他们。公平一点讲，与这些“狂热分子”相比，不可理喻型的人还要好一些。总体来说，不可理喻型虽然缺乏理性，但他们还有实际的问题。不可理喻型的人之所以不想尝试新语言，可能是因为他们自己不喜欢你推荐的那种语言的语法。而“狂热分子”不想尝试新语言，则是由于他们深信桌面出版将是你所在行业的未来，因此除非基于纸面的解决方案重新流行，否则他们将一直拒绝新事物。

两者之间的区别，我个人认为是理论上的。实战中，对付“狂热分子”与对付不可理喻型的人没有什么不同。不过要注意的是，“狂热分子”确实存在——确实存在！



11.3　几句忠告

如果实际情况完全如我所说，那么你不可能说服不可理喻型的人。无法说服是一回事，但不一定等同于他们不会参与实施你的建议。你可以对他们投其所好，迎头痛击，软硬兼施。实际上，对于这一类人，我建议的总体策略是忽略他们，让管理者批准使用你推荐的技术。

这里的关键在于，你无论如何也不可能说服他们。所以，没有必要在他们身上花过多时间；否则，徒劳无益。

Part 3 第三部分

技　巧

本部分内容

第 12 章

装满工具箱

接下来这几章介绍一些应对的技巧，你可以用它们来对付那些怀疑者，把他们引导到你的思路上面来。其中大部分应对技巧都不仅局限于某一类怀疑者。同样，你通常也需要综合运用某几个技巧才能起到作用。

其中一些技巧是通用的。换句话说，这些技巧在大多数情况下都会有效果，每一种技巧都有其独特的“内在价值”，也完全是你力所能及的，而且它们也不依赖于任何有利的外部条件。这些技巧包括：

- 取得经验；
- 传达理念；
- 展示技术；
- 建立信任。

其他的技巧可就不完全取决于你了。除了你自己的意愿，是否使用这些技巧还要视某些外部环境而定。比如说，公之于众确实很有效果，但如果涉及的代码关系到公司的知识产权，那恐怕你连共享一行代码的权利也没有。问题的关键在于是不是有使用它的时机，有则用之，用就有奇效，比起前面那几个通用技巧管用多了。这些技巧包括：

- 适当妥协；
- 公之于众；
- 注重合力；
- 搭一座桥；
- 来点刺激。



为了在实际工作中运用这些技巧，别忘了把那些对怀疑者行之有效的技巧列出来。先尝试位于前面第一个列表中的技巧，因为这些技巧仅仅需要你个人有意愿即可。然后，再找机会使用前面第二个列表中的技巧。



第 13 章

取得经验

千万不要推荐别人使用那些自己都不熟悉的工具或技术。只花一个下午的工夫翻阅一些FAQ，然后就得出某种工具适合你们公司的结论，那是远远不够的。孤陋寡闻型人的不打算自己调研某种技术，而百般挑剔型的人则不会让你轻松过关，激情燃尽型的人可没有那么容易忘记你介绍的那种技术曾报废了他们整整一周的工作量。面对如此局面，你必须得用自己对所选工具的知识来反驳他们的说法，当然也得知道这种工具的不足之处。



可是，要想精通任何一种工具或任何一门技术，没有几个月甚至几年时间是不现实的。我并不是说为了推荐某种技术或工具，得要等自己成为相应领域的专家再付诸行动，而是说一定要朝着成为该领域专家的方向去努力。除了广告词里宣传的那几个特色功能之外，你还得知道这种工具什么情况下、什么地方存在局限性。假如你是个新手，那必须得熟悉它才行；如果你已经是个熟手了，那就进一步贯通；如果你已经贯通了，那就更上一层楼。关键在于要主动获得一些专门的经验，而不是在那里裹足不前。

部署的故事

Ed遇到了似曾相识的麻烦。今天的午餐会上，大家谈着谈着，话题又转到了部署上面。现在的部署简直就是一场噩梦：操作规程、备忘清单、手工测试，即便不出差错，每次没有半小时也下不来。三次部署差不多总会有一次出问题，导致项目的公共用户界面在半小时的修复周期内只能以500错误示人。每次出问题以后，接踵而来的通常都是一次领导约见，当然不会有什么好事。尽管大家都觉得这种局面难以接受，但却迟迟没人愿意改变。

Ed曾倡导过使用Ant来代替手工部署。他以前在自己的几个个人项目中尝试过Ant，也知道它的长处和不足。事实上，他还专门针对团队的项目做过一次概念验证，使用了Ant构建、测试和部署脚本，结果证明是可行的，每次运行只要45秒。他再次跟团队成员们提到自己的验证。

一位同事，Bernard，打断他说：

“我也装过Ant，也运行了。可是在我运行你的脚本时，发现无法运行FTP任务。不能用FTP，还谈什么部署？”

这个问题Ed是知道的，他自己也遇到过。他为此还在安装指南里专门写清了如何解决该问题。

“这个问题其实挺常见的。你得保证Jakarta ORO jar包和commons-net jar包都在你的classpath中。这些我在安装说明里都写过了。我再给你发一份吧。”

Cynthia也开始唱反调：

“我看了很多博客文章，都说Ant已经过时了，Maven比Ant好用，关键是Maven不用XML。”

Ed对此也有准备：

“首先，Maven也使用XML，所以我就不明白你在讨论的究竟是不是Maven。其次，Maven在构建过程中要求遵守很多约定，这些约定跟我们的做法不相符。当然，我也知道Maven更符合极客的口味，但是对我们特定的流程来说，还是采用Ant才能更快地上手。”

Herbert也接着说：

“我在上一次任务中使用过它，可我当时认为你们大家都觉得它有问题。现在好了，我希望能用它。”

Umberto已经归顺了：

“我以前不知道Ant和Maven。但看样子Ed已经做了不少这方面的研究了，假如我点一个按钮就可以在45秒内完成部署，那可真要谢天谢地了，还有什么好矫情的呢？Ed，你抽空教教我怎么用吧。”

Ed点头答应。这次的讨论结果是三比二。公共界面再次对外停止响应，领导或许得听听Ed、Herbert和Umberto的意见了。尽管停机事件并不是他们造成的，但他们已经有解决方案了。



13.1　技巧分析

挑选新的技术或工具是个苦差事。即便新的技术或工具将来能够节省时间和精力，但总还是有点远水解不了近渴的意味。为了推广新技术，要么需要有积极性，要么需要有捷径可走。或许你很有积极性，别人都不如你的兴趣浓厚。但这并不是说别人有什么错误，只不过他们并没有你那么要求上进罢了。既然他们没有这种进取心，你就得成为他们的捷径。

回忆一下你在学习自己准备推荐的工具时付出的种种努力。有一次你发现它不能正常运行，可能是因为自己不知道该用什么语法，也可能是忘了某个选项，而在你打算通过Google搜索来找到解决方案时，却又不知道该搜索什么。这件事记忆犹新吧？那时候，你有两个选择：继续或放弃。你选择了继续，因为你对这件事一直充满激情。一定要让你的同事在面临类似情形时，能够再多一个选择：问你。



在前面的故事当中，Umberto之所以愿意试一试，就是因为Ed答应找时间教教他。这种帮助别人的能力和意愿是让孤陋寡闻型的人改变立场的关键所在。此外，通过迅速、有见地、有理有节地化解Bernard的问题和驳斥Cynthia的说法，Ed也表露出领导Herbert和培训Umberto的信心。

13.2　怎样成为专家

要掌握你所推荐的不同的工具或技术需要经历不同的过程。但这也不是说学习这些工具和技术就没有一些通用的途径。以下就是取得经验的一些途径。

13.2.1　研究技术和工具

研究技术和工具的沿革，搞清楚发明它们的初衷。换句话说，就是要搞清楚它们是什么时候、在什么地方、为什么、由谁，以及怎么创造出来的。还要知道存在哪些竞争性技术，要确定这种工具到底适不适合你的组织。要清楚它为什么合适，而另一种技术为什么不合适。在前面的故事中，Ed显然已经对Maven的情况了然于胸。他知道Ant及其竞争性技术的长处和不足，

因而能让Cynthia无话可说；甚至，他利用了Cynthia犯的错误，更让自己具有了权威性。

只有过程，没有终点

在我跟某些人说到要通过研究工具和技术成为专家时，他们一点信心都没有，好像还被吓了一跳。他们并不认为自己可以在某方面成为专家，觉得这个要求对他们而言是不可能做到的。但事实并非如此。

技术经验，包括那些极为特殊的技术经验，都不是一成不变的。除了那些已经过时的技术，你所要推荐的任何工具、任何技术或者任何产品无不日新月异。产品的功能会随着时间推移而增加。开发社区会发布更多的注解。用户也发现Bug，而开发团队会修复这些Bug。一切都处于变动之中。从这个意义上说，即使是那些最有经验的专家，也只能说他们对最近接触到的某个版本有经验。专家以及他们的经验都不是静止不变的。如果两年不关注新的技术动向，往日的专家就会变成今天落伍的人。



由于知识的更新速度太快，真正的专家与过气的专家之间的区别就在于能否持续不断地学习和更新。既然连专家都要不断学习和更新，那么真正的专家与要努力成为专家的人之间就没有太大的差距。只不过那些专家在这条路上比你走得稍远一些而已。

记住这一点，就可以随时保持自信和谦逊。不能与时俱进的专家就算不上是专家，因为只有与时俱进才能让人成为专家。

13.2.2 实际使用

任何工具或技术都其局限性，而每个人都有一件或一堆处理不好的事儿。不管是设计师、开发人员，还是市场营销人员，都不会在网站首页公开他们的那些麻烦事儿。通过FAQ页面或支持论坛或许能找到一些问题的线索，但如果不狠下决心亲自尝试，永远不可能知道它在你的环境中会出什么问题。

对Bernard提出来的问题，Ed之所以能那么快地给出答复，就是因为他经历过同样的步骤，亲手安装了Ant。他知道FTP所需的JAR文件没有包含在标准的产品中，为了避免自己的同事在同一个问题上不知所措，他还专门为此写了文档。只有那些有切身体会的人，才可能做好这种准备。

在选择作为试验对象的项目时可要小心点。如果技术或工具是普适性的，而你正在与一些可能会反对它的人合作某个项目，那就不要用这个项目试验。否则就会在自己作好准备之前暴露该工具，降低人们对它的期望值。

在前面看到的那个故事里，Ant作为将手工任务自动化的工具不会带来多少风险：因为它只是已有流程的替代品，而且也只有Ed使用它。就算是对本地代码的版本控制，也不算什么大问题。可是，如果是重写整个应用程序，让它遵循某个特定的设计模式，而其他团队成员又都不熟悉这种设计模式，那恐怕就很难让大多数人接受了。

13.2.3　向现有专家求助

寻找熟悉你想推荐的工具或技术的专家，与他们建立良好的关系。如果当地有个该技术或工具的用户组，一定要加入该用户组。如果没有，还可以通过论坛或博客找到其他专家。



尝试着从这些专家那里学会避免一些新手常见的错误。了解他们经历过的一些因为错误使用该工具或技术而导致的“惨剧”。跟他们学习有助于帮你掌握应对将来的新用户的各种知识。

13.2.4　教人使用

教人使用是获取经验中比较靠后的一个步骤。没有什么比向别人传授相关技术或工具的知识更能让人增长才干的了。自己使用工具或技术，无非就是想方设法让自己能顺利地用起来而已。对那些已经没有感觉的工具来说尤其如此——我们不会多想一点，也不会真正理解它到底都做了什么。但在教别人使用的情况下，仅仅向人家描述几个笼统的步骤是远远不够的；你必须要跟别人解释清楚每一步都做了些什么。

当然，最难的还是找到想学习的人。而这个人最好不在你希望他们采用该项技术或工具的团队里面。如果在外部实在找不到人，那就在团队内部挑一个孤陋寡闻型的人，而且这个人不大可能变成百般挑剔型、激情燃尽型或不可理喻型。

13.3　适用对象

取得经验之后，可以有效地对付以下几种怀疑者。

13.3.1　孤陋寡闻型

所谓孤陋寡闻型，顾名思义，就是指他们缺少有关某种技术或某方面的知识。只要你成为专家，那就等于具备了他们所缺少的大量知识。你知道的越多，就越容易推进。

第二点也非常重要：你应该充分共享自己的知识。否则，你很难让孤陋寡闻型的人真正服气。因此你必须做到百问不厌，悉心教导，不断激励他们。付出总会有回报——孤陋寡闻型的人是最容易说服的，如果你在其他怀疑者那里要花两分钟或三分钟，对他们只要一分钟就够了。



如果这一点还不足以鼓舞你的斗志，那我就不得不重申：教会别人是最好的学习方式。要想解释清楚某项技术的工作原理，必须思考和理解得更多，而不仅仅是会用。教会别人能让你更像一位专家，从而让你能更好地向别人传授知识，从而让你可以教给别人更多的东西，从而能让你更像一位专家……用计算机术语讲，这就是一个无限循环；而在哲学里面，这个过程叫做正反馈循环。

13.3.2　随波逐流型

随波逐流型的人必须有人出来领导他们。如果连你自己都不确定要怎么做，何谈领导别人？只有知道你的工具能做什么，以及——更重要的——它对你的组织意味着什么，才是领导那些随波逐流型的人的关键所在。

有没有领导能力还要另说，关键得看随波逐流型的人愿不愿意让你领导。你对自己工具的掌握程度给他们带来的信心，是鼓舞他们跟随你的一个重要因素。

13.3.3　百般挑剔型

百般挑剔型的人不能用习惯性思维来理解。他们过去曾经对某些技术有

过指望，但无一例外都失败了。他们是大学里总向教授提一些讨厌的问题来展示自己的聪明人。于是，他们会挖空心思想出一些刁钻的问题来阻挠你的推广工作，以避免自己因变革而怅然若失。要想让他们尽早闭嘴，你就得以充满自信的语气和入木三分的解答来回应。

关于百般挑剔型的人，还有一个重要的事实要注意，那就是他们一般都博而不精。这个一般化论断还是有根据的。假如一直都不考虑使用什么新技术或新工具，那就不会去尝试。所以，百般挑剔型的人通常只知道皮毛而已，抓住这一点，就很容易让他们闭嘴。他们会跟你提到一些竞争性技术，或者讲一些耸人听闻的故事。但是，只要你的发言比他们更有深度一点，他们就会乖乖地知难而退了。

13.3.4 激情燃尽型

激情燃尽型的人使用过你推荐的技术，但却没能用好。以成为专家为目标，取得对某种技术的经验，能够让你更好地理解到底是什么让激情燃尽型的人心灰意冷。只有切实了解他们的遭遇，才能更好地回应他们的问题，甚至还有可能得体地指出他们自身存在的某些问题。



得体，没错，这一点很重要。一旦让人觉得你是在攻击他们，或者是在不客观地批评他们，那反而会促使他们更加坚持自己的立场。客观不客观当然不是你说了算，他们自己会感觉到。所以，一定要记住，要成为专家，而不要成为“万事通”。

13.4 陷阱

在使用本章这个技巧时，你可能会遭遇两个陷阱：

- 在学习过程中，你就强迫他们去使用；
- 在推销过程中，让人感觉傲慢或霸道。

13.4.1 强迫别人

大多数人在学习一门技术的时候，都会先找一些简单的示例，或者一些测试代码。由于这些示例或代码都不复杂，所以大多数工具都能顺畅地运行。

我们知道这一点，因而就会想在实际的工作中试用该工具或技术。此时此刻，你会尝试在自己的项目代码中使用它们。但在此之前，别忘了先冷静地想一想，这样做会不会迫使别的项目成员也要跟着你一道使用它。对某些技术来说，这样做就一定会迫使别人也跟着你试用，而你呢，并没有准备好向大家展示。

在前面的故事里，Ed只是将当前的一个流程用Ant来自动化，其他人不必因此做任何改变。假如从未用过某种MVC框架就准备转换到该框架，那么就是另一个极端了。这样的转换，无法限制在自己负责的部分而不影响其他人。

当然，这都是些显而易见的情形；真正有挑战性的则是介于这两者之间的情况。就拿单元测试来说吧，乍一看它不会影响到别人，因为你只针对自己编写的代码编写单元测试。可是，在开始写单元测试以后，你就会改变编写要测试的代码的方式。一般来说，这样会驱使你把代码写得更出色，但却有可能不兼容以前写的那些代码，抑或由于别人没有认识到这一点，会对你写代码的方式感觉到不舒服。



为此，我建议你把学习试验的范围限制在自己的独立项目内。你不可能知道自己的学习和试验会对他人产生什么影响。你的目的是要说服、转变他们，而不是把他们都变成激情燃尽型的人啊。

13.4.2 傲慢或霸道

从和蔼可亲的专家到高高在上的“万事通”，这中间很可能只有一步之遥。区别在哪里？归根到底就看你能否倾听别人的意见。“万事通”只听自己的——只要他想出一个方案，就没有别人说话的余地了。不管可行与否，这种人都会口若悬河、滔滔不绝。而专家呢？真正的专家会认真倾听每一个人的声音，并在倾听的基础上找到解决方案。

有些时候，丰富的专业知识很容易让人感觉到一种霸道。或许你一开始就以专家的身份自居，认为自己始终都是正确的。但这种态度会给人留下你好像很有优越感的坏印象，进而影响你的工作。总而言之，你得分清这个因果关系：因为你通常是正确的，人们才会给你贴上专家的标签；而不是因为你是专家，所以你就是正确的。

13.5 小结

取得经验对于说服那些最容易转变的类型而言，不失为一种非常有效的方式。而且，说服这些人也是转变所有怀疑者的重要一步。但要想说服更多的人，只凭这一种技巧还不够，还必须结合其他一些技巧。

除了你作为布道者向团队推荐自己相中的工具或技术之外，取得经验还有另外一个好处：你可以利用自己的业余时间来学习，也可以抽出一部分工作时间来尝试。这种学习和尝试无需他人的参与，而且将来也能为你的简历增添几分光彩。

简单点说吧，取得经验有很多优点，风险又很低，应该是你的首选。无论如何，我都希望你尽可能成为自己所推荐的工具或技术的专家。没有了这个先决条件，后面要讲到的其他技巧恐怕就都是浮云了。



几点建议

怎样才能迅速地积累经验呢？下面我就给出几点建议，供参考：

- 通读你所推荐的工具或技术的完整手册；
- 找到相关工具的论坛，看看自己能不能回答别人提出的一些问题；
- 就你想推荐的工具和技术写一些博客。

第14章 传达理念

好了，你已经找到了一种技术，它比同事们正在用的技术更好。这种技术更好，所以这个理由就足够了，是吗？可惜的是，更好和足够好并不是一个概念。如果程序长在树林深处，一直不崩溃，那会有人知道它吗？不会。先抛开这个诡异的比喻不说，如果你不把这个工具告诉别人，别人永远也不会想到用它。

你得向别人宣传自己想要推荐的工具。在宣传这些工具的同时，还要牢记两点：

- 不要吓跑听众；
- 要吸引听众。

没错，首先必须考虑不要吓跑他们。无论你自己觉得多么有吸引力，但一吓着他们，他们就会跑掉。就算你想要推荐的工具再好，没人关注、没人重视，又有什么用呢？

版本控制的难题

John发现了Git[①]，而且已经为之着迷了。可以说，Git的每一个功能他都非常喜欢，就连那些他自己也不十分清楚有什么用的小功能也不例外。总之，他就是喜欢，喜欢得都有点过头了。

Patrick则是CVS[②]的老用户了，他当时正考虑换一个版本控制系统，因为CVS的限制实在让人无法忍受了。为此他也做过一番研究，发现

① 一种分布式的版本控制系统。要了解更多信息，请参考*Pragmatic Version Control Using Git* [Swi08]。

② 一种集中式的版本控制系统。要了解更多信息，请参考*Pragmatic Version Control Using CVS* [TH03]。



SVN是最合适的一个，虽然差别不大，但还是要更好一些。SVN那句“把CVS做对”（CVS done right）的口号，可以说正中他的下怀。他已经做好转换的准备，但遇到了John。

“如果你考虑的还是CVS，好又能好到哪里去呢？”John喊道。

Patrick开始维护自己的立场：“不管怎么说，我的那些工具都使用它。”

“废话！我们不都使用Eclipse嘛，Git也能在它里面用。而且，它运行得更好，还是分布式的，也就是说……”John嘟囔着。

此时此刻，Patrick的心思已经不在这个问题上了，他只是一味地点头，直到John说完。Patrick也就此打消了升级源代码控制系统的念头。

几个月后，又有人向Patrick提到Git。这一次，人家只是大讲特讲Git的优点，丝毫没有揭SVN的伤疤。Patrick试了试，结果他也喜欢上了Git。如果John当初没有那么激进……



在前面这个故事当中，John面对的是一个对改变充满激情的人，而他却给人家当头泼了一盆冷水，浇灭了人家的激情。更重要的是，John不仅失去了一次让Patrick认识Git的机会，还导致了Patrick裹足不前。这个故事的意思就是：不要像John那样。不要把人吓跑；要把他们吸引过来。说起来容易，做起来难。下面还是来讨论一下该怎么做吧。

14.1　技巧分析

人，就算是搞技术的，归根结底还是人。是人就有不理智的时候。我们有时候会根据事实做出正确的判断，另外一些时候，我们也可能会被某些事物华丽的外表所迷惑。在技术领域中，优秀的技术何止千万？正是因为很多人只看表面，才导致那么多优秀的技术始终没有流行起来，比如Betamax、Smalltalk、FireWire。

究其原因，就是因为人们在很多情况下，会被感情所支配，无法做出理智的决定。事实上，人们感情用事的时候比真动脑子的时候更多。

所以，包装才这么大行其道，表达也成为一门学问。不过别担心，没必要非得告诉市场部那些家伙你也认同这一点，就把它当成我们的一个秘密吧。

14.2　掌握表达的艺术

很多人都以为，与人交谈、联系和施加影响这些事，你要么会干，要么就不会干。而且，有些人擅长与人交往，有些人则不。但事实并不是这样的。没错，有些人天生就比较擅长与人交往，也有这方面的天赋。可到头来，怎么跟人联系还是看有没有技巧，这种技巧是可以学会、可以掌握的。

14.2.1　要做人，别做计算机

程序员们会花很多时间盯住计算机屏幕。正如尼采曾指出的："当你凝视深渊时，深渊也在凝视你。"你已经拿出了自己时间中可观的部分与各种机器在一起生活。作为程序员，我们会在一开始以二进制的方式来看待这个世界：如果我推荐的东西没错，那我就是对的，而这也应该是其他人在评断我的方案时的唯一标准。遗憾的是，这种情况鲜有发生。人是一种感情动物，其他程序员也不例外。即便你真的正确，仅仅告诉人家他们的选择不对也还是不够的。



所以，不要告诉人家他们现在的这个选择"不对"。不要好像人家误入歧途一样跟人家谈话。事实上，如果不是特别必要，就别提人家的当前状况。那谈什么呢？谈你的工具如何管用，如何高效。

14.2.2　要有激情，但不能激进

激情与激进这两者的区别很微妙，不过我想还是可以像下面这样来描述。

- 激情，是指你深深地喜欢一种技术或工具，希望其他人都来用它，因为它可以让大家把工作做得更好，而且能简化工作，提高效率，最终会使工作变成一种享受。你承认在有些时候确实需要其他方案，但你也知道自己的工具在那些情况下一样不会逊色。
- 激进，也是说你深深地喜欢某种技术或工具，认为所有人都必须使用它，理由就是它更好。你想象不出来什么情况下不应该使用它，而一旦遇到这种情况，你就会找出种种奇怪的理由对这种情况大加责难。

这两个定义之间的界限也不是十分清晰。但总之，如果你所说的话全都围绕一定能改进自己同事的工作，那么你基本上就是有激情。如果你所说的话全都围绕让大家接受正确的*开发方式*，恐怕你就有点激进了。

关键在哪里？关键在于不要强调你的工具好还是不好、正确还是不正确，更不要牵扯正义还是邪恶。不要把话说得太满，也不要声称自己的工具适合任何情况，尤其是在还有其他更好的替代方案的情况下。

14.2.3　要提建议，而不是申饬

很多时候，在听说别人还没有使用我们推荐的技术时，我们的第一反应就是去吓唬他们。

- 为什么你不用Git呢？
- SVN？明摆着是个错误。
- 你应该马上就迁移到Git！



当然，在你跟别人这么说话的时候，你并不想让人们都躲开。这也许只是你对内心那份热爱的一种宣泄。但别人并不这么认为。你这么一说，只会激起别人反驳的欲望。要想鼓励人去做什么，更好的办法是提建议，而不是申饬别人。

- 你考虑过Git吗？
- SVN？我用SVN总是麻烦不断。
- 我在很多项目上都用Git，都很好。

14.2.4　要多听，而不是多说

命令别人去做什么事很难成功。因为他们会维护自己的立场。即便是随波逐流型的人，也是希望能得到领导，而不是被别人强制。更好的做法是问他们几个问题。

- 为什么你会选择它呢？
- 你想通过它来解决什么问题？
- 它能不能很好地适应你的工作流程？

紧接着是一个比较难处理好的环节——提了这几个问题之后，要认真听他们怎么回答。然后，才能理解他们的背景，明了他们的问题。就因为他们感觉到有人认真听，所以他们会给你更靠谱的回应。此外，你们之间的争论如果都能围绕着他们的问题展开，那最终的效果还会更好。

怎么确定自己是不是认真地听了呢？用自己的话把他们的回答复述一遍，问："我理解得对吗？"你可能会觉得这么问显得自己在装腔作势或者傻乎乎地，但是一定要克制自己，过这一关。然后，他们就会对你的认真倾听表示肯定。多做几次，你就会习以为常。

14.2.5　保持积极的心态

想一想民主竞选的活动，消极应对恰恰是示弱的表现。发表所谓的竞选活动不公平之类的言论没人会相信。说你的工具比竞争对手更好是更好的策略。而说你的解决方案会让追随者的日子更好过，工作效率更高，但又不作出担保，则是推销的最高境界。



但是，这并不意味着你不能与竞争性产品或技术做比较，或者诚实地说出你对竞争方案的感受。不过就像Patrick Swayze在电影《旅馆》（*The Road House*）中扮演的角色那样，无论什么时候，都要尽量让人感觉你很友好。

14.3　适用对象

传达理念这个技巧对以下怀疑者行之有效。

14.3.1　孤陋寡闻型

孤陋寡闻型的人是很容易转变的。但正如他们容易转变为支持者一样，他们也照样很容易变成其他类型的怀疑者。对他们而言，你需要抓住任何一次机会。对这种类型的人，最糟糕的做法就是你太过强势，导致他们走向反面。

14.3.2　百般挑剔型

百般挑剔型的人可不仅仅是喜欢跟你争吵，而是需要跟你争吵。虽然不

可能完全不让他们否定你的意见，但你可以少给他们一些可乘之机。传达理念就是消除这些可乘之机的一种方式，而且还有可能让他们听你的。

14.3.3　不可理喻型

实际上，你不可能真正影响不可理喻型的人——他们有自身的本质问题。因此，无论多么务实的解释，都不可能打动他们。然而，你的解释能够保证让你自圆其说，能够让人听起来合情合理。这样一来，如果不可理喻型的人再对你说三道四，就会让人觉得他们求全责备、吹毛求疵了。

14.4　陷阱

你可能认为把道理讲清楚不会有什么负面作用，大多数情况下确实不会。但是，从开始使用这个技巧起，你就可能会让人感觉有点假惺惺的，甚至会让人感觉你成了产品推销人员——那就更差劲了。所以，务必时时刻刻小心谨慎，待人以诚，千万不要**范式**啊什么的满天飞。



14.5　小结

本章的核心思想在于，怎么去传达，比传达什么更重要。还有一点也很关键，不管你说了什么，他们只相信自己听到的。更进一步，回忆自己过去与别人交谈的情景，与这两点对照一下，然后再想方设法改进自己的表达方式。

几点建议

怎么才能更好地传达理念呢？下面我就给出几点建议，供参考。

- 练习带着感情来表达观点，可以找一位志同道合的朋友，或者干脆自己练。关键是要声情并茂。
- 练习时要录音，然后放给自己听。如果你不认真地听自己说，那还谈何信心？
- 如果你把自己的想法写成了邮件，那么在发送之前先搁置一小时，然后在点击发送按钮之前再读一遍。

第 15 章

展示技术

小学的时候，有一次上讲故事的课，我至今记忆犹新。记得当时老师对我们大叫："表演，不要光说。"后来，我二十几岁，就参加专业的即兴喜剧演出。导演还是对我喊："表演，不要光说。"今天，我想对着你们也喊一次："表演，不要光说。"

这是什么意思呢？下面我们来看两种可能的故事开场白。

- John走在大街上。John很伤心。
- John在大街上踽踽独行，想到与心爱的Lisa失之交臂，他的眼里不禁泪光闪闪。

这两句话传达的意思一样，但第一句只是告诉你简单的事实，第二句则向你展现了一幅画面。哪句话更有可能吸引你？哪种开场更有故事性？

虽然推荐专业的开发工具时不可能像讲爱情故事那样，但背后的思想是一致的。要想让某人了解一种工具，它的功能、优势以及使用方法，最好的方式莫过于把它展示出来。

电视马拉松编码

Ed写过一个代码生成机。在这个生成机的基础上，经过改进，他又构建了一个代码生成器。给它一个数据库，它可以利用数据库内省、ORM以及公司自有的UI组件，在不到一秒钟的时间内，就构建一个特别棒的应用。这个生成器很灵活，灵活到可以在生成代码之后，还可以再修改，这样就可以在对生成的代码没有破坏性变更的前提下控制业务模型。简言之，这就是Ed一直以来梦寐以求的大师系统。



但老板Bob，对此并没有那么理想主义。Bob理解的是这个生成器能生成代码，而Ed则说它能生成很棒的代码，可他的解释和说明也就到此为止了。Ed无法说服Bob让他在工作时间来做这件事。

那天是周五，Bob和Ed在讨论开发一个工作任务跟踪系统。Ed想了一下这个问题，说他的代码生成器能够胜任这项工作。Bob否定了他的这个想法，因为他不知道构建这个系统需要花多少时间。

Ed回到家，凑巧的是，周末电视会播他最喜欢的警察与本地检察官之类的情景剧。他拿着遥控器，抱着笔记本，把自己埋在沙发里头就开始干活了。他用自己的生成器创建了应用，又花几小时完善了UI和业务模型。

周一又到了，Ed把生成的应用拿给Bob看。

Bob有点难为情，他问："你是怎么用一个周末的时间写好这个应用的？"

"就是用那个你看不上眼的代码生成器啊。"Ed回答。

Bob愣了一秒钟，然后央求道："快，再给做一遍，从头开始做。"



这个小故事表明，无论Ed跟Bob说多少次自己的工具能做什么，Bob都不可能真正明白。只有看到它的实际应用，看到它所解决的问题，以及看到Bob想象不到的开发速度，才知道到底是怎么回事。好，Bob现在都看到了，而且这一看就给他留下了深刻的印象，因为他的想法被以创纪录的时间变成了实际的应用。

15.1 技巧分析

人常说，百闻不如一见。确实如此。还有一句话，叫"不见棺材不落泪"，说的也是这么回事。可能是因为人们之间缺乏信任吧，他们如果看不到激动人心的演示，而只听见你念叨一大堆功能，就会扭头离开。甚至，由于私心的作用，让他们看到自己同样面临的问题，要比只给他们看一些含含糊糊的假设的方案更有冲击力。

15.2 把握展示时机

展示要考虑时机。你可以默默地做好充分准备，只待时机成熟；也可以

主动创造展示的机会。这两种把握时机的方式各有利弊。自然而然的机会没有规律可循。或许是某人主动问你在哪方面比较有经验，或许你正在研究某种技术，而他们自己上门求教。换句话说，不是你要让他们听，是他们要你讲。当类似这种机会出现时，人们更倾向于接受你的说法，但你没办法控制特定的状况。创造机会当然就是你主动去寻找机会。或许是你发起一次自带午餐的研讨会，或者在某个特定的时间讲一讲你的工具。在以类似方式创造的机会中，你可以完全控制局面，但也更容易听到质疑的声音。当然，你可以等待机会，也可以创造机会，或者相机而动。这些都取决于你，取决于你准备得是否充分，取决于你发现机会的眼光，取决于你等待机会的耐心。

15.2.1　等待机会

在前面这个故事里，Ed在等待机会出现。他已经有所准备了，而当机会一出现在他面前，他就抓住了它。有些工具和技术只有通过自然而然的机会才更可能被人接受。



源代码控制工具就是这样一个例子。要想创造一个能够展示源代码控制工具从失败中恢复代码的机会是很难的。当然，如果你真的创造出这样的机会，就怕到头来不得不考虑换一份新工作了。好吧，在某个倒霉的家伙弄出点乱子来之前，你就只能耐心地等待那一刻。等那一刻一到，你只要一两句命令行调用就可以把代码恢复过来。这个等待的过程可能会比较长，然而在你展示之后，大家一定会心服口服。

15.2.2　创造机会

从另一方面说，即便是那些最适合在自然而然的机会中展示的技术，也不妨在人为制造的场景中展示一番。源代码控制不仅仅只在灾难恢复中有用，它还能用于追踪频繁的发布。向大家展示一下这个功能，尽管可能不如失败恢复那么震撼，但照样也能吸引人。

要创造这样的机会不难：把大家叫过来，展示给他们看。至于怎么展示，展示什么，完全取决于你。你可以把它当作一次工作时间的午餐交流。或者你来组织并安排一次季度的展示会。大多数公司都有这种制度安排。

15.2.3 培养机会

还有一种等待加创造的混合方法，这种方法叫做培养机会。你可以设置一些“诱饵”，吸引某个人“上钩”，然后抓住机会做一次有准备的展示。

就我的职业而言，我是一位软件布道师。我为自己想要传道的工具准备了各种各样的代码示例。可是，如果只是拿起来就演示，经常会发现听众们注意力并不集中。而如果我可以让谁要求我演示一下那个工具的话，大家马上就会集中注意力。我怎么做呢？有大量不同的方式，不过在写这些文字的时候，突然有一个想法在我脑海里蹦了出来。我把自己准备推荐的软件的标志做成商标贴，贴在了笔记本上。当有人在咖啡馆里看到一个商标贴，问我这是什么的时候，我立即抓住这次机会，开始布道。我向他们讲述这些工具，然后他们还想进一步了解。于是，一次自然而然的机会就诞生了。由于这种情况没有强制的成分，所以大家一般都会抱着求知的态度认真听，而我呢，我是准备好了的，所以就能够最大限度地说服他们。



15.2.4 代码评审

最后，还有一种创造机会的方法，可以让你展示你看中的语言、框架和编码技术，那就是代码评审。你可以向别人展示一下使用你的工具完成同样的任务有多么容易。代码评审还有其他可以利用的地方，相关内容将在其他章再讨论。就目前来说，可以通过代码评审引出对其他技术的展示。这样，就好像这些并不是你非要给大家推荐的某种专业开发技术似的。

15.3 适用对象

展示技术这个技巧对以下怀疑者非常行之有效。

15.3.1 孤陋寡闻型

孤陋寡闻指的是缺少某方面的知识。通过展示技术，就可以暂时避开功能介绍，而直接演示该工具能帮他们做什么。这么简单的一次试用，很可能就让他们茅塞顿开。

15.3.2 百般挑剔型

百般挑剔型的人喜欢争论。你可以跟他们讨论观点、功能、承诺，但如果没有实际的结果摆在面前，只凭空洞的描绘，还是很难获胜。事实胜于雄辩，展示技术能帮你打败他们。

15.3.3 时间紧迫型

对于时间紧迫型的人，展示技术可以让人事半功倍。你可以在很短的时间内向他们展示你的技术的效果——注意，关键是很短的时间。展示的时间虽短，但对他们而言，已经足以看到潜在的好处了。

15.3.4 不可理喻型

不可理喻型的人会千方百计拒绝你的好意。而在别人成功展示某种技术的应用之后，再毫无道理地反对，那就只能说明他们自己不可理喻了。正是这种担心暴露的心理会让他们对你敬而远之。虽然不能让他们听你的，但至少可以不让他们成为你的绊脚石。



15.4 陷阱

在依靠展示技术说服别人时，也存在一个显而易见的风险：墨菲定律[①]。现场展示技术有可能不成功。一个接一个地演示，说不定哪一个就会让你在众人面前下不来台，导致展示彻底失败。要缓和这种局面，也有一些方法。比如，你可以预先准备一些能用的代码，然后当场录入。也可以切换到一张表明整个过程最终成功了的屏幕截图。可不管怎么做，结果反正都是失败了。就像演示成功能说服别人认可你的技术一样，演示失败也能说服别人否定你的技术。

演示失败并且补救措施也不管用的时候，最恰当的做法是停下来。停下演示之后，你要从容地跟大家解释，不要让人察觉你内心的慌乱。这样稳住

① 墨菲定律（Murphy's Law）：事情如果有变坏的可能，不管这种可能性有多小，它总会发生。——译者注

自己或许不能给自己的演示结果加分，但至少你不会再失分。此时此刻，不再失分就是最好的结果，必须得接受。

15.5 小结

展示技术是适用范围最广的一个技巧。这个技巧可以拿来对付很多类型的怀疑者，而且耗时短，见效快。如果你能提前做好充分准备，在某个自然出现或刻意创造的机会到来之际，成功地展示出你的技术，那么说服别人的效果将是非常好的。

两点建议

怎么才能更好地展示技术呢？下面我就给出两点建议，供参考：

- 就你想推荐的技术写一个演示，讲给自己的同事听；
- 把你对工具的演示录成视频，通过博客分享出来。



第16章

适当妥协

公司成立时间越长，规章制度就越多。这些规章制度通常都是在发生某些意外事件之后制定的，目的就是避免同样的事件再次发生。规章制度一多，其弊端就显露出来，往往导致约束多于保护。规章制度还存在另一个主要问题，即一旦颁布，鲜有更改。只有极少数公司会反思一些制度是否还有存在的必要。很多时候，由于技术的成熟，某些规定也会变得不合时宜。结果，人们只能因循守旧地按照一些过时的制度行事，而这些制度所要预防的威胁是不可能再出现的了。这种情况会极大地影响士气，但同时也是一次推动变革的机会。



折磨人的存储过程

Jeff所在公司的SQL管理员有一条规定：所有数据库活动都必须在存储过程里完成。为这件事苦恼的不只Jeff一人，所有开发人员无一幸免。

几年前，他们公司的网站曾经被人以SQL注入的方式攻击过。所谓SQL注入，就是在提交的表单附加一些SQL命令，寄望于从不恰当的输入处理过程中找出漏洞。有点类似于在银行支票后头加上一句“再提100万美元”之类的。（为了避免出现这种情况，我们都会在自己开的支票上画一道线。）

防止SQL注入的办法不止一种。比如，培训开发人员，让大家头脑中紧绷着安全这根弦。而Jeff他们公司采取的方式，则是强制所有代码都要通过存储过程，以保证DBA能够全部检查一遍。此外，还强制使用参数化的数据库通信方式，这一条更有效地防止了SQL注入。

Jeff所在团队的大多数人都不喜欢写存储过程，也不愿意因为写那

些八股文式的SQL语句而浪费时间。而最让他们愤愤不平的是连修改一个简单的select语句都得通过DBA。可是，Jeff 每次建议大家使用某某ORM工具，队友们又都不同意。其中一些人反对确实是因为不想用，而另一个非常主要的阻力还是源自DBA不会允许在应用中生成SQL。

另一方面呢，DBA们也是牢骚满腹，抱怨着自己每天有多忙。一点儿也不奇怪，对每一条操作公司数据库的SQL语句都要进行检查，能不忙吗?

Jeff 的团队主要使用Java开发，所以Jeff 知道Hibernate是一个比较合适的ORM方案。他对Hibernate的工作原理也进行了一番研究，发现Hibernate对大多数操作都使用参数化的查询。

Jeff 把这个发现告诉了队友，也告诉了几位DBA。于是，队友们接受了这个方案，因为大家可以不用再被迫去写存储过程了。而DBA团队也同意试一试，但条件是必须在SQL Profiler运行的同时进行试用。

几个月后，所有新项目都用上了Hibernate。DBA们要检查的CRUD SQL语句少了很多。而开发人员也很高兴，因为他们不用再无谓地使用存储过程了。大家都觉得这是一个进步。



在前面这个故事当中，Jeff 拿出了一个兼顾DBA和开发人员的折中方案。结果，这个双赢的解决方案不仅把不合时宜的限制性规定扫地出门，还让大家采用了他推荐的工具。大家每天的工作也都因此轻松了许多。

16.1　技巧分析

你是在令人痛苦的陋习和新的工具或技术之间找平衡。试一试，如果让人们在一个痛恨的东西和一个不了解多少的东西之间进行选择，那他们会怎么做？他们至少会听一听那个他们不了解多少的东西是什么。你是在利用人们对某一规定的憎恶来推动变革。可以肯定，我们会获得推动变革的感情分，但你必须把握好各方的诉求。

16.2　找到折中方案

要利用折中方案推销你的工具和技术，必须要找到一个条件成熟的规

定，作为提出折中方案的机会。这条规定呢，起码得所有人都痛恨，都抱怨，都对它一肚子怨气。在找到这条规定之后，接下来就要创建合适的方案，以便让你的工具终结那条规定。

16.2.1　找到条件成熟的规定

条件成熟的规定一般都是众所周知的。是不是所有开发人员都在拿一个规定开玩笑？他们会不会在每次面对这个规定时都怨声载道？但愿能有这么一条规定，它还能允许你变通。

还有一些可用的规定并不那么明显。或许原先导致制定这条规定的意外事件令人极其恐怖，你的那伙开发人员被震慑得只有乖乖就范。或许由于大家对这条规定还不是非常清楚，所以尚未体会到它的不合理。

以下是几种适合变通的规定。

- 绝对化的规定：不允许例外情形的规定容易成为众矢之的。
- 安全相关的规定：人类对于涉及安全的意外事件容易反应过度，因此会制定一些针对所有人的规定，有了这些规定不仅不会让人感觉更安全，反而是更受限制。
- 行业最佳实践：有些人会直接把行业最佳实践作为公司的规定，而不管它是否适用或有必要。
- 外部强加的研发规定：如果非开发团队的人也可以对如何开发应用或者如何沟通指手画脚，那他们定的那些条款很可能合适。
- 没有理由的规定：任何规定的背后都应该有一个明确的原由。这样开发人员才可以说“因为Y，所以我们要X”。如果没有人知道Y，那么这些规定恐怕就值得商榷了。



16.2.2　找到与规定匹配的技术

在找到可以利用的规定之后，紧接着就是去找与之对应的技术。前面的故事展示的场景比较直截了当。要求使用存储过程，就是为了防止SQL注入，而这样一来就给开发人员和DBA造成了工作压力。Hibernate可以降低SQL注入的威胁，同时减少工作量。这种情况下的技术选择比较简单，因为都没

有脱离数据库操作这个范围。

下面我们再来看一个没有那么直观的例子。假设老板定了一条规矩，要求对Web应用的每一处修改，都必须经过他亲自测试才能发布。这个规定会导致一些问题。老板本人不可能有那么多计划之外的时间。Web应用开发团队的更新速度也会随之降低。而在老板不在公司的时候，大家甚至都不知道应该做什么。总之，这个规定是亟待改变的一个机会成熟的规定。于是，开发人员建议使用Selenium（一个HTML UI测试框架），这样他们就可以自己编写测试并将测试自动化。如果老板愿意的话，他也可以自己写测试。不过这样一来迭代速度就加快了，而且这样测试也比老板的随便点击来得更严谨。一般来说，用户验收测试与自动化测试是两码事，但在当前的情况下，一个框架可以同时实现两个目标。



16.3　适用对象

适当妥协这个技巧对以下类型的怀疑者行之有效。

16.3.1　时间紧迫型

制度意味着约束。时间也是一种约束。时间紧迫型的人，他们的压力就源于各种各样的约束。扫除制度性的约束等于节省了时间。一方面节省了实施规定的时间，另一方面节省了检查规定落实情况的时间。替这些人节约时间，可以让他们对你刮目相看。

16.3.2　发号施令型

事实如此，我们也不得不面对：提到规则，开发人员的定义是“最佳实践”，而管理人员的理解则是“规定”。表达方式上的差别，足以体现大家的思维上的差异。管理人员是规定的推行者。他们要监督很多人，因此那些对所有人都适用的规定执行起来会更容易。要想让他们废除某条规定，唯一的办法就是拿东西跟他们交换，然后才可能让他们有所放弃。毕竟，规定对他们而言，就是他们要面对的问题的解决方案。只有给他们提供更好的解决方案，才能让他们心甘情愿地放弃老的规定。

16.4 陷阱

在考虑使用这种技巧时，有一个非常大的挑战，那就是并非所有公司都存在能被技术革新完美取代的规定。前面举的两个例子尽管也来源于现实，但类似情况我见到的还确实不多。这就属于机会不多但值得把握的。

另一个挑战，就是要团结一批有类似想法的人。大家拧成一股绳，到时候能够将相应的规定一举推翻。通常，规定执行者跟你的团队越疏远，转变起来就越困难。如果某些规定只是开发团队自己内部的约定，那是最简单的。如果这些规定是由人力资源部门定的，那可就要看运气了。在这两种情况之间，难易程度高低不一，需要你具体情况具体分析，以便拿出有效的应对方法。

16.5 小结



适当妥协并不像其他技巧的适用面那么广，但在合适的情况下，效果却非常之好。这样说吧，不要总是指望着有使用这个技巧的机会，而一旦机会出现，就要大胆尝试。

两点建议

怎么才能更好地做到适当妥协呢？下面我就给出两点建议，供参考：

- 向团队成员了解哪些规定或者编码标准最让他们头疼；
- 时不时地就要想一想“我们公司/团队是否还需要这条规定”，即便对那些答案明显是“需要”的规定也要定期反思。

第17章

建立信任

摆事实讲道理对于说服别人考虑你的意见至关重要。但是，一个经常被忽视，而同样非常重要的方面，就是信任。假如没人相信你，即便是推广最简单的工具和技术，你都可能陷入孤军奋战、无人响应的境地。

与其他技巧相比，建立信任相对要困难一些，它没有现成的公式可以套用，也没有既定的步骤可以遵循。我们可以讲一讲如何取得经验，但如何取得别人的信任就没有那么简单了。事实上，取得信任差不多约等于多干好事，少干坏事。不过也不必害怕，尽管没有捷径可走，可也不是说没有办法得到大家的信任。



FUD因子

Shailaja的公司要花钱买一套新数据库。因为预算实在太高，她正在寻找一种可能的替代品。在经过深思熟虑之后，她还是决定把公司的系统都换成MySQL。从她的角度讲，MySQL很经济，技术支持也很靠谱，而且相关的信息很透明。

很多人都反对Shailaja的决定。有些人纯属耍无赖，他们对老技术有一种惯性，不过有些意见还是有道理的：改变需要成本；而这个成本可能是过高了。而且，即便目前从各方面来讲MySQL都很合格，但MySQL刚刚易主，未来如何谁也不能确定。

讨论已经到了白热化的程度，突然一个人脱口而出："我听说开发我们用的老系统的公司，明年就准备关门了。"

这句话的效果很明显，所有人都动摇了。但问题在于，Shailaja知道这个消息不真实。一些唯恐天下不乱的人在某个论坛里散布了这个制造FUD的谣言，而那家公司一直都在辟谣。

FUD

FUD，代表 Fear（害怕）、Uncertainty（不确定性）和 Doubt（怀疑）。尽管这个短语是 20 世纪 70 年代中期才被发明的，但这个概念可谓源远流长，大概可以追溯到一个原始人向另一个原始人卖石头，“以防那些大怪物再回来”。近来，它又成为营销人员、公关宣传人员以及推销员们对用户进行恐吓的工具。说白了，就是给你讲一堆关于竞争产品的烂事儿，让你望而生畏，从而掏钱买他们的东西。

技术人员的圈子里或多或少也存在这个现象。那些使用专有工具的程序员，会传播一些关于开源工具许可中暗藏玄机的“秘密”。而开源支持者呢，则四处讲述某些专有工具中包含隐藏代码的“恐怖故事”。

这些谣言最终都会被戳穿。谣言骗得了一时，可骗不了一世。所以，靠谣言获利不会有好果子吃。随着人们不断被蒙蔽继而看穿这种把戏，就会有人站出来公开事实真相。真相一旦传播开来，这种伎俩将不再有效。



她也很为难。看到这种FUD给自己的同事带来的恐慌，自己心里其实还是挺想含糊过去的。可是，这个消息的的确确是误传。她知道，一旦真相大白，大家都会因为被戏弄而暴怒。但他们发现真相的概率很小——因为这些人从来都不上那个论坛。

最终，Shailaja还是澄清了谣言，FUD也烟消云散。MySQL呢，也还是被采用了。而大家之所以同意了她的决定，其中一个原因就是大家都认为她做事向来光明磊落、从不欺心。

17.1 技巧分析

建立信任之所以可行，是因为人们都不喜欢被操纵。甚至，人们都不希望感觉到自己被别人操纵了。这就是为什么二手车推销员在美国的文化中有那么根深蒂固的不良形象。他们被视为欺诈和操纵的代名词（尽管事实经常并非如此）。

在更换工具或技术的时候，你经常会发现所有的经验都拿出来比对过了，所有的选择也都细细地斟酌过了，而此时对于到底该选择哪一个大家仍

然莫衷一是。在一个公司面临这个门坎的时候，通常都需要有人打破僵局，最终一锤定音。这个一锤定音可能并不是靠理性，也不是靠什么经验。某种程度上，最终的决定源自对一个人的信任。如果你能得到大家的信任，那在决策过程中遇到了类似的坎儿时，其价值不可估量。

17.2 如何建立信任

前面我已经说过了，建立信任这件事没有什么灵丹妙药。从长期的角度看，倒是有一些对建立信任至关重要的事情需要注意。要是说这些事你无论如何都必须做，那恐怕会引起一些争议。毕竟在这样的问题上，还是仁者见仁，智者见智的。

17.2.1 不要故意撒谎

故意撒谎很容易被人识破，因为谎言一听就是谎言。像什么“MySQL不是关系型数据库，因为它不处理外键”之类的话，就是故意撒谎。这种说法非常直截了当。如果有人发觉，你可以说自己说错了。但是，假如你经常这么玩儿，终有一天会被人揭穿。



幼儿园老师都会告诉小朋友，好孩子不能撒谎。而之所以我在这里还要对你们说不要撒谎，是因为我们会为自己撒谎找到种种借口。特别是在谎言中还带有那么一点点真实性的情况下，比如前面关于MySQL的那句话。MySQL在默认情况下不支持外键，但是在把表类型设置为InnoDB后，它就支持了。这也是人们容易迷惑的地方。类似这样微妙的问题很容易成为谎言的借口。总之，就是不要有意撒谎。

17.2.2 不要回避事实

回避事实并不像故意撒谎那么容易被发现。在某些情况下，你没有必要说出一些假话，只是不讲出某些真话，而这些真话在对话过程中又是至关重要的。比如，像“MySQL的表名区分大小写”这样的话就属于回避事实，因为它回避了“在某些操作系统中”这个必要条件。如果你在参与讨论从旧数据库向MySQL迁移之类的话题，那说明这个问题显然是非常重要的。

回避事实的危害不会像故意撒谎那样大，因为人们会倾向于把人往好处想，他们会认为是你搞错了，或者是你一时疏忽。而这也正是它的吸引力所在——这种“说谎”方式，好处大，风险小。不过，这样的情况一旦多起来，你在人们心目中的印象无非就是以下两种之一：

- 你有意地漏掉了某些细节，因此靠不住；
- 你总是弄错，因此也靠不住。

无论人们对你得出哪种结论，都对你的信用不利。

17.2.3　永远不要制造FUD

从本质上讲，FUD是有一种破坏性的手段，因为它是通过恐惧胁迫人来作出决定的。而人们迟早会知道事实真相。一旦他们知道了事实真相，每个人都会挺身而出，检举揭发。散布FUD被人揭穿之后，特别是当FUD都是凭空捏造的情况下，你几个月乃至几年的信用都会因此搭进去。有的人从此以后就再也不会相信你了。



即便人们并没有察觉你是在操纵他们，他们也可能会认为你是那个“高喊狼来了的孩子”。如果你口口声声预言的那些灾难、问题还有厄运都没有兑现，人们就会认为你的话不可信。这跟建立信任可是背道而驰的。

不过，这也确实带来一个问题。“如果我不能散播FUD情绪，那又让我怎么讨论竞争性的技术？从某种角度说，难道我不是要说其他技术或工具不好吗？”

对此，我想说，任何时候都要从自己的工具说起。其他工具并不是不好，只是你的更好罢了。如果有人让你说说竞争性工具的缺点，那你就真得说说了。不过，即便说，也要说得清楚明白、不掺杂个人感情，更不能言过其实。此外，如果可能的话，可以提前准备一些业界专家对这些竞争产品的评论，说不定能用得上。

17.2.4　承认错误

也许听起来不太可能，但你也肯定会犯一些这样或那样的错误。极端情

况下，你甚至从一开始就是错的。可能你会下意识地对这些错误视而不见，或者会有意地转移自己的注意力。在你努力想让别人采用某种技术的时候，这种诱惑还会变得更加无法抗拒。

无论你信还是不信，犯错误对你来说是再好不过的事情了。因为这样你就有机会告诉大家你错了。对，就是这样。听我说，大多人都不习惯别人直截了当地指出自己的错误。即便能接受，多数人还是必须得面对承认自己的错误这个难题。不过，如果你能欣然承认自己的错误，那大家就会信任你。原因是你尊重客观事实，即便说出来对你自己不利。

如前所述，如果你的错误与工具或技术有关，那就需要重新评估。承认自己搞错了而又执迷不悟，才是彻彻底底的大傻瓜。

17.3 适用对象



建立信任这个技巧对以下类型的怀疑者行之有效。

17.3.1 激情燃尽型

对于激情燃尽型的人来讲，他们对你所说的一切都有自己的经验，而你所说的一切与他们的经验不符，因而他们很难相信你。假如让他们感觉到一丝一毫的不真实，他们将立即守口如瓶。不要给他们任何借口。

17.3.2 百般挑剔型

百般挑剔型的人，不管别人说什么，他们都会质疑。给他们一个怀疑的理由，只会让他们拿出双倍的精力去证明你就是他们想象中那样的骗子或傻瓜。千万不要真的骗人，不要授人以柄。

17.3.3 不可理喻型

不可理喻型的人就是想找一个借口，好阻止你推行什么技术变革。如果你成了骗子，对他们来说简直就太好了。他们会在后续的工作中，用你的诚信问题来攻击你。这个局面可不是你想要的。

17.4 陷阱

说真话没有多少危险。但有一点倒是挺值得注意：在努力建立信任的过程中，你会指出并强调自己提出的方案中的不足。你想告诉人们自己推荐的工具有哪些不足、缺点或者危害。但是，向大家宣扬这些问题是另外一码事。

可惜的是，对此谁也没什么好办法。因为没有东西是完美无缺的，偶尔你也会被人问及，你推荐的工具或技术有什么缺点。那就说呗，把缺点说出来，不过别忘了讲清楚为什么你认为这些缺点与自己方案的优势相比算不了什么。

17.5 小结



表面上看，鼓励做人要诚实似乎有点荒唐可笑。但是别忘了，现实当中诱惑人说谎的因素比比皆是，特别是为了一些短期利益。从长期来看，说谎的代价实在太大了。说出事实真相吧，超越短视的行为，放眼长远吧。

两点建议

怎么才能更好地建立信任呢？下面我就给出两点建议，供参考：

- 在自己的头脑中想象那么一种情境——竞争性的技术比你打算推荐的更合适；
- 搞清楚自己的解决方案到底存在哪些弱点，以及这些弱点会在什么情况下暴露出来。

第 18 章

公之于众

如果一棵树在树林中倒下，而树林中空无一人，那么这棵树倒下的时候是否发出声音根本无所谓，因为没有人能听得到①。没错，公之于众就是要让人们看到你的工具或技术，但仅仅看到还不行，还要向他们证实你的工具或技术有用。为此，你要让公司之外的人——不管是专家还是同行，选择你的工具和技术。有时候，这样做会比单纯向同事们推荐效果更好。



全局Bug跟踪器

Jim已经厌倦了自己公司项目中使用的那种跟踪Bug的方式。这种方式实在太简陋了。用户在碰到某个错误后，要给技术支持写邮件，但他们在邮件中不一定会附上错误消息。即使有些用户在邮件里附加了错误消息，仅凭那点儿信息也很难追查到问题出在哪里。

Jim在自己的应用程序中添加了一个全局错误处理功能，这个功能可以把错误和应用程序的状态及栈跟踪信息放到一起发送一封邮件。他把这个功能写成了一个新的Bug跟踪系统，能够捕获并记录公司环境中的错误信息。实际上，他写了一个非常棒的定制的Bug跟踪器。

可想而知（你们都已经看过本书前面那些章了），他的队友们都不想使用这个Bug跟踪器。尽管这个系统使用起来更方便、效率更高，也容易搜索，甚至还有其他很多好处，但他们就是不用。

Jim当然不想让他的成果变成森林中那棵无声倒下的“树”。于是，他把自己的代码作为一个开源项目公之于众了。工作中碰到类似困扰的

① 西方流传着一个关于树的哲学故事，大意是：如果森林里有一棵树倒下了，而周围又没有人迹，那么树倒下去有没有声音？事实上，这个故事与禅宗的“境由心生”不谋而合，犹言“无心即无物”。——译者注

人们很欢迎它。其中有人开始尝试Jim的这个跟踪器，甚至还有人帮着增加这个系统的功能。几个月之后，Jim发展出一个人数不多但全球化的用户社区。

这时，消息传到了Jim老板的耳朵里，说世界很多地方的开发人员都在使用Jim的工具。老板开始过问这件事，询问为什么公司内部都没有用。管理层最终统一了认识，要求在公司内部使用。

18.1　技巧分析

《圣经》里提到过一种模因[①]，即“在自己的家乡永远成不了先知”。[②]基本意思就是，你知晓了一个非常重要但又极具争议的事实，但那些了解你调皮捣蛋历史的人是不会听的。因此，你必须到自己家乡以外的地方去布道，才能得到信任和尊重，才能有追随者。



同样，工作中也莫不如此。你的同事们也许还在谈论你刚到公司两周的时候，有一次在代码里埋了一个Bug，结果搞得整个系统都完蛋了。每当你建议做什么而有人要反对时，他们都会翻出这件事来打压你。

总之，就是因为太熟了，所以才会导致轻视。人们通常都会拒绝来自公司内部的东西，也不能接受自己身边的人一下子“牛”起来。通过将你的工具和技术公之于众，让外界来验证它们的价值，可以解决这个问题。他们对你有偏见，这个工具是你写的或者你打包的，那你就无法说服他们。可是这些陌生人呢……他们是不会骗我们的。

希望你从我的口气里能听来出，我也觉得依靠这种方式有点可悲。我们之所以能加入公司，是因为我们有能力胜任自己的工作，而且我们也具备与工作要求匹配的知识积累。我们可不应该依靠这种方式。但是，有时候不依靠这种办法还真不行。

---

① 模因（meme），源自英国著名科学家理查德 · 道金斯（Richard Dawkins）所著的《自私的基因》（*The Selfish Gene*）一书，指“在诸如语言、观念、信仰、行为方式等的传递过程中与基因在生物进化过程中所起的作用类似的那个东西”。具体到这句话，就是指下文中所说的那种人类普遍存在的思想观念。——译者注

② 《圣经》里的这句话，还有其他译法，比如“先知在自己的家乡是从不受欢迎的”、“没有先知在自己家乡被人悦纳的”。总之，意思大概相当于我们常说的“外来的和尚会念经”。——译者注

18.2　让自己成为焦点

这可不是说让你雇个宣传员到外面去给你做宣传。你得在技术社区里面得到认可。以下是几条行之有效的途径。

18.2.1　开源你的工作

把你的工作成果开源，是赢得大家认可的一个最好方式。可能会有很多人都在试图解决遇到的类似问题，开源就能得到他们的关注。SourceForge、GitHub等这些大的源代码托管站点是很好的选择。通过这些代码分享站点，你可以给自己的项目分类，同时加上详细的文档。而且，很有可能会有人向你伸出援助之手，帮你完善代码。

不过，在你迫不及待去做这件事之前，还需要再考虑几个问题。你的开源项目即便没有人使用，管理起来也是颇费时间和精力的。写文档、注意代码可读性、确保可移植，这些全都是你在开源之前必须准备就绪的。如果有人参与进来了，你还必须处理好他们贡献的代码、论坛话题以及Bug报告。



你可能需要获得许可

我在这一章建议你把在内部推荐受阻的软件开源。不过，根据行业和公司的不同规定，恐怕你得在开源之前先得到公司的许可。很多公司对员工在工作时间取得的劳动成果的知识产权问题上有明确规定。

不过，别因为有规定就不敢去问。我就曾在一个大型研究机构和一个大型软件公司工作，这可是两家典型的具有知识产权保护传统的组织。但他们都准许我开放源代码。当然，每个人的实际情况会有所不同，所以在开源之前别忘了查一查员工手册。

此外，还存在一种危险，即你可能模仿了一个已经开源的项目，因此你自己的项目无法引起别人的注意。对于这个问题，你应该事先花点时间，做一番市场调查，看看是不是已经有了雷同的项目。当然，也可以到经常去的技术社区里面问一问，看看到底有没有人需要它。

这些问题不一定会直接打消你开源自己项目的积极性，但确实不应该只凭一时冲动就去做这件事。如果你的项目非常适合开源，那你可以通过发布它来将它公之于众。如果不是这种项目，那无非就是赔上时间和精力，去创建了一个无人问津的开源项目。

18.2.2　参加竞赛

技术圈里的许多供应商、社团或者出版社，都会举办一些竞赛。这些竞赛往往会围绕着让主办方和参赛者双赢的宗旨来组织。主办方最后可以拿到演示程序、示例代码、白皮书以及用例。而参赛者则可以赢得奖品和知名度。

要参加比赛，必须拿出点真本事来证明自己。无论是使用一些特殊的功能，还是展示一些特定的问题，只要你的工具或技术可以胜任，就完全可以用它去帮你攻城掠地。然后，只要谈到入围比赛，就可以顺便提到你的工具和技术。



显然，这样做的风险就是你可能会干投入而收获不到知名度。这就要取决于竞赛规模以及其他因素了。大多数竞赛都会努力宣传以赢得关注，但假如你碰上了1000位竞争者，那恐怕一场硬仗是免不了的了。不过话又说回来了，竞争越是激烈，荣誉也会越高，怎么选择你自己看情况定吧。

18.2.3　为得奖而设计

奖项也是由举办比赛的同类组织授予的。不同之处在于，他们接受已经发布的作品。因此就存在白忙活一场风险——万一你做的项目别人早就做完了呢？

不过，大多数评奖活动都有日程。发起方会有一些想要推广的东西，而他们想要一些相关的实例。他们可能会把条件公布出来，也可能不会。因此，成功的把握也许不如参加竞赛那么大。然而，失败是无伤大雅的，所以我说，不如放手一搏！

18.2.4　让人评审你的项目

公之于众，并不仅限于到公司外面去。也可以想办法在公司内部引起注意，比如走一走正式或非正式的代码或项目评审的门路。评审通常都需要一

些总结性的信息，利用这个机会，你就可以给大家讲讲自己的故事，讲讲你的工具到底帮了多大的忙。接着，你可以再展示一下在它的基础上构建起来的代码或者项目计划。这样既可以让你的成果公开化，而且还会按照你的章程行事。

不过，关键还在于会议的同步性。无论是大家坐在一起面对面，还是远程会议，抑或其他什么形式的电话会议，都没问题。如果把会议变成了异步的，比如发邮件呀，或者写维基啊什么的，也就失去了借助会议形式讲故事的优势，毕竟只有开会才可以让大家在同一时间内把精力集中在一件事情上。我们要认清这个事实，假如人们会认真看每一封邮件、每一个维基页面，那你就不用为如何吸引注意力费心了，因为大家早就已经知道你都干了些什么了。

18.3 适用对象

使用公之于众这个技巧对以下类型的怀疑者行之有效。

18.3.1 孤陋寡闻型

无论如何，只要能吸引孤陋寡闻型哪怕一点点注意力，都有助于他们向了解你的工具或技术迈进一步。为此，可能只要确保让他们真的注意到你就够了，至于提醒他们注意你的合理方式，也许是发一封参赛或获奖的邮件。

18.3.2 百般挑剔型

对于百般挑剔型的人，获得外部的认可绝对能够收到先发制人的效果，因为连“外人”都认为你的工作有价值，值得尊重。不过，百般挑剔型的人很可能会贬低你的成绩：“谁不知道，那个竞赛的水平很烂啊。”你没有什么好办法阻止他们这样做，但却需要做好准备还击。

18.3.3 激情燃尽型

施展这个技巧可能会有一批人称赞你的工具或技术有价值，也可能会有一批人因你的工具或技术而取得成功。不管怎样，都将是对激情燃尽型的人以往经验的一个有力否定。



18.3.4 发号施令型

前面不是说过嘛，这些工具或技术通常并没有在管理人员考虑的范围之内。要求他们任何时候都能从技术优势的角度来认识你的贡献，是不太可能的。可是，从另一个方面讲，理解“获奖方案”就容易多了。

18.4 陷阱

别忘了还有一些比较现实的考虑。显然，如果你打算推荐的工具或技术本身就来自外部，那就无法使用这个技巧。如果你在推荐Subversion，你不能拿它去参加任何竞赛，或者把它以开源形式发布。换句话说，只有你自己开发的工具或技术才适合。

出于种种原因，你们公司可能不愿意让你随便把在公司内部开发的系统向外人公开。另外，这里面还有运气的成分：光是开源，或者只是有这个打算，并不意味着你就真的受到公众关注。说到底，这也是一种赌博——你投入的时间与可能的回报相称吗？这个问题你必须得想清楚。



18.5 小结

公之于众是影响身边人，尤其是管理人员的非常有效的策略。它能够战胜人们对内部开发的系统的偏见。但这个技巧也不是在什么情况下都合适，必须是你自己的解决方案才能公之于众。

几点建议

怎么才能更好地做到公之于众呢？下面我就给出几点建议，供参考：

- 搞清楚你的工具是否还有其他竞争性的开源方案；
- 弄明白你的公司对开源自己的工作成果有什么规定；
- 为参加的“竞赛”和相关的技术领域创建一个Google Alert[①]；
- 同样，也为“评奖”和相关的技术领域创建一个Google Alert。

---

① Google Alert（http://www.google.com/alerts），是Google通过主题搜索和邮件通知帮助用户跟踪事件进展的一项免费服务。——译者注

第 19 章

注重合力

尽管我们每天都活在技术里，但除非你是在一家技术公司，否则技术不会决定你们公司的方向。你所在的公司处于一个行业中，这个行业有自己的规程，也有自己特定的问题。除了行业特定的问题之外，还会有一些通用的问题，比如安全、浪费或者环保等所有公司都要面对的问题。有时候，需要通过技术手段来落实行业规程，解决特定的问题。所有公司也都会承受法律制约及可能的行业制裁。如果你的工具或技术能够用于解决其中的某个问题，那无异于获得了相当强的推动力。

Spring更上一层楼

接上峰指示，鉴于最近恐怖主义猖獗，Argon Electronic所在行业每种产品的任何零件的销售，都必须事无巨细地以特定格式记录到一个特定的地方。目前用Java写的这个库存管理系统无法满足这个要求。

Markos曾经推荐过使用Spring。这个框架能解决与公司业务相关的很多问题。他个人主要指望着通过该框架来管理依赖注入。迄今为止，他已经取得了一些进展。

然而，Spring AOP支持面向方面编程，你可以包装对象并在之前和之后拦截方法调用以运行代码，而无需改变底层对象。这对于管理**横切关注点**或需要在应用程序各个地方重用的代码集合非常有用。这可不仅仅是个巧合——记录日志，那可是使用Spring AOP来管理横切关注点的经典范例！

管理层认为要解决这个问题必须得进行一次大规模耗时间的重写。Markos的方案表明，通过使用Spring AOP外加几个日志类，只要远少于



预期的时间就能把问题搞定。Markos想在项目中使用Spring的计划有了进展，而他也把Spring领进了公司的大门。

19.1　技巧分析

我们这些搞技术的人可能不愿意相信：技术不会驱动商业；商业只由商业驱动。商业问题始终都比技术问题更重要。把自己的工具和技术与商业需求联系起来，你就能创造出让别人使用它的比较充分的理由。此外，你通常还会因此得到管理层的注意、关心和保护。这样，让你的同事采用你的方法也就具备了更重要的背景和更强烈的动机。

19.2　构造合力

可以用来构造合力的手段并不是太多。机会要么存在，要么不存在。但是，你可以提前做好准备。



首先，大多数诸如此类的规章都要花很多时间去贯彻落实。很少有一夜之间就改天换地的可能。因此，在你订阅行业门户站点新闻的同时，不妨留心一下身边都有哪些机会出现。

其次，尽量把自己想要推荐的工具或技术与更大一些的问题联系起来。这里存在安全隐患吗？这个安全隐患是不是已经严重到可以把安全作为突破口的程度？

最后，听听管理层的声音。他们说过哪些问题是明年要重点关注的？你们是不是要面临一次财务危机了？你的技术能否给公司节约一笔开支？你们是不是准备快速地扩张？你的技术可以让扩张更容易吗？

19.3　适用对象

注重合力这个技巧对以下类型的怀疑者行之有效。

19.3.1　时间紧迫型

能够借助形成合力来解决的问题，一般来说都是无法回避的。换句话说，

这一类问题的关键就是必须解决。对于管理层或者规章制度所宣布的变更，任何人都不可能视而不见。无论人们是否反对，公司都会安排时间来执行这个变更。这正是你跟时间紧迫型的人讨价还价的基础。他们已经接到指示，要为这件事投入时间。关于是不是有时间的问题，已经有结论了。现在，问题是你要给他们提供一个解决方案，让他们能在相对长的时间段里面节省一部分时间。对于争取时间紧迫型的人而言，没有比这个机会再合适的了。

19.3.2 发号施令型

不言而喻，这个策略是专门为管理人员量身打造的。这是把你的工具变成解决他们问题的方案的最佳例子。对于管理人员，最紧要的问题是，你必须把自己的工具或技术设计得让他们一眼就发现它是一个合适的解决方案。别忘了，关键在于商业上的因素，例如成本、时间、服从和工作，而不是诸如性能、封装等这一类的技术因素。



19.4 陷阱

构建合力是一种环境造就的策略，对于环境的变化你是无法掌控的。你不可能自己把规章制度中的某一条改掉，为自己创造机会。只有当客观条件具备的时候，这个技巧才能用得上。

即使让你逮着了这么一个机会，也免不了会有人将它视为儿戏，或者抱着某种侥幸心理而有意加以抵制。这些不过是人类的本性而已。但是，这些问题你都可以控制。不是有一句老话吗，叫做："如果你手里拿着一把锤子，那看什么都像是钉子。"因此，一定得保证自己的技术真的跟商业问题有着恰当的结合点。可不能干那种削足适履的事儿，明明不合适非得生拉硬拽，强求一致。请参考第3章来判断自己的工具或技术是否合适。

19.5 小结

本章介绍的技巧是十分管用的，但前提是必须等机会出现。关键在于要时刻记着：公司的存在都有一个理由。这个理由通常不是技术。如果你能把自己的工具与该理由联系到一块儿，那么就不仅可以获得管理层的帮助，而

且可以直接为公司的最大利益服务。

几点建议

- 编制一张包含行业相关规章制度的表格，看看其中有没有你的技术能够对应的条款。
- 确定你的技术是否可以跟下列通用的敏感问题扯上关系：
    - 安全；
    - 财务浪费；
    - 环保。



第20章 搭一座桥

有时候，某种工具或技术与大家已经习惯使用的工具或技术相比差别确实太大。要想让人们完全接受它，实在是太困难了。在完成一个任务的过程中，人们情愿容忍的变化程度是有一定限度的。也许你的工具或技术是另一个平台上的一种新语言。这就意味着要经历两次转变，不是一次。要想让公司从现在使用的技术过渡到你推荐的技术，不可能一步就跨过去。在这种情况下，我建议你设法搭一座桥。



这里所谓的“桥”是一种过渡性的工具或技术，不是你心目中最终的解决方案，但也不是现在使用的。搭这座桥的目的在于，先使用一段时间过渡性技术，待大家适应以后，再转换到最终的解决方案。

有时候，可以找到一些现成的过渡性方案。这种方案类似于“针对Y的X”，例如“针对.NET的Rails”，或者“针对Java的代码隐藏”，等等。这些过渡性技术确实存在，如果能够善加利用，千万不要错过，它们能为你节省非常多的时间。然而，在没有现成的过渡性技术时，你就必须要考虑自己动手，创造出自己的解决方案来。

不仅仅是一个框架

我刚刚发现了一个针对ColdFusion的框架，它的名字叫Model-Glue 2。这是一个非常典型的MVC框架，而且集成了另外两种非常流行的框架（依赖注入及ORM）。所有这些框架就构成了一个“平台”，通过它可以基于数据库来生成应用。这些特性对所有语言以及任何开发小组来说，都是相当标准的配置。

然而，我的那些同事对转换到这个“平台”都不怎么热心。究其原因，主要有以下几块绊脚石。

- 我们公司有规定，所有SQL语句都必须包装在存储过程当中；这就让采用ORM方案变得不太可能。
- 这个框架属于主动型生成器，因此会存在性能限制。但这仅仅是人们的一种先入为主的感觉。其实，这个框架在配置当中可以用一个产品标志来提高性能。然而，默认安装要显示一句“Hello World”都要花两秒钟，这就给人一种难以抹掉的坏印象。
- 鉴于使用其他框架的负面体验，人们通常会倾向于对当前这个框架产生偏见。
- MVC在这个公司里并不是所有人都喜欢的模式。

于是，我就把重点放在了Model-Glue最让我欣赏的方面：“平台”。我着手编写为我实现这一点的代码生成器。这个生成器写了几个月，它的基本运行原理如下。



- 分析数据库并为每一个表生成CRUD存储过程。
- 然后，检查存储过程以生成模型组件（相当于Java中的类）。
- 接着再基于存储过程生成UI。
- 通过每个表一个的控制器将各种CRUD视图连接起来。这并非真正的MVC模式，但比我们原来使用的更接近MVC。
- 以上每个步骤都是被动完成的，因此实际的应用不会因创建操作导致消耗。

这个生成器有一些优点。它没有违反工作中只能使用存储过程的约定。由于代码是被动生成的，因此也就没有所谓性能的问题。同样，我从来也没有称它为“框架”；相反，我管它叫“代码生成器”，这样做一些反对者可能就会给它一次机会。

几个月后，公司中的每个编程组都至少在一个新项目中使用了我的代码生成器。更可喜的是，有些小组还在尝试Model-Glue，因为这些小组对它的认识已经慢慢地发生了转变。

20.1　技巧分析

之所以会有这个技巧，是因为改变不容易，尤其是大的改变要比小的改变更难。作出多次的小改变往往要比一次性的大改变要容易。然而，多次小改变所花的时间不一定比一次大改变所花的时间更长。大的改变通常需要牵扯更多人。大的改变需要得到一级又一级公司领导的批准。参与的人和管理层次越多，需要的时间也就越多，加在一块，可能会比把同样的变化分成多个较小的改变来实现的时间更长。

之所以会有这个技巧，是因为你了解自己的小组。你知道自己公司里那些人老发什么牢骚，你也知道大家最痛恨什么问题——这些问题就是让你下定决心搭一座桥的关键。在前面的故事里，不得不把所有数据库连接代码连接起来完成CRUD是大家所痛恨的。我们的桥可以解决该问题，同时又不会强制大家为改变而付出多大代价。换句话说，我们的过渡方案把利益最大化，把代价最小化了。



20.2　搭一座桥

搭一座桥的意思非常明显。你得琢磨琢磨自己的解决方案和公司的实际状况，并找出让大家难受的地方。然后，要么创造一个，要么使用现成的“桥”来解决问题。

20.2.1　情况调查

你必须要观察一下自己所处的环境，还有你的解决方案，然后弄明白这两者之间冲突的关键在什么地方。然后，你还得搞清楚解决方案的哪个方面对于你的组织是最迫切需要的。

假设你正努力向一个开发小组介绍TDD（Test Driven Development，测试驱动开发），而该小组此前从未做过单元测试。除非这个小组主动想采用，否则他们永远不可能使用TDD——因为这里的差距可不是一星半点。但是，假如你们公司的Bug修复工作经常相互冲突，那你就有机会针对Bug修复来引荐单元测试了；但不要强制实施TDD。因为单元测试可以解决大家面临的

一个难受的问题，同时又不必强迫大家花时间学习TDD。等到单元测试推行成功之后，可以再尝试TDD。相对于使用单元测试之前，这个小组在那时会更容易接受它。

20.2.2 找一座桥

一种解决方案在某个平台上流行起来之后，经常会被移植到其他平台。不信可以搜索一下Hibernate in .NET、Rails for Java，或Spring for Python。很多这样的“桥”其实已经存在，而且是已经写好了。在这种情况下，你要做的只是从已有的方案中选择一个。这些已经存在的过渡方案并不是为你的公司定制的，但你可能也不需要定制。

20.2.3 搭一座桥



在本章的故事当中，我没有在自己的方案和现状之间找到一座现成的“桥”，因此我就自己写了一个。所谓搭桥，无非就是把最终的方案改造一下，以便让它适合当前的环境。如果你们有一些约定或者最佳实践，你还得保证自己的方案与它们一致。如果所有应用都有样板代码，要记得包含它们。总之，就是要尽可能在自己的过渡方案中加进各种有利的因素。既然都是你自己设计并开发的，何乐不为呢？这样一来，唯一的缺点就是工作量会很大。

20.3 适用对象

使用搭桥技巧对以下类型的怀疑者行之有效。

20.3.1 随波逐流型

随波逐流型的人需要被领导。搭一座桥有助于把自己放到领导的位置上。说这个方法对随波逐流型的人是立竿见影的一点儿也不夸张。

20.3.2 百般挑剔型

百般挑剔型的人肯定会对你的方案说三道四。这是客观条件。但是，如

果你是在定制一个过渡性解决方案，那就不要急着还击他们，先妥协一下。他们不喜欢这个方面，那就改。强迫他们去做什么事不太容易；因此，给他们提供选择。当然了，你不可能对他们批评的所有方面都做出改变，但你可以给他们一个容易接受的结果。

20.3.3　激情燃尽型

激情燃尽型的人，跟百般挑剔型的人类似，也会因为你能够修改既有方案而有所转变。无论他们以前有什么经验，你都可以针对他们的要求定制啊。如果某个特殊的约定或模式曾在过去导致他们遇到过麻烦，你就不要用。围绕他们过去的痛苦经验，体贴地将这些额外的定制需求考虑进去，能够明显改变他们的态度。

20.3.4　时间紧迫型



时间紧迫型的人最经典的说法，就是他们喜欢尝试一些更好的东西，只是没有时间。因此，你早晚会碰到他们，甚至他们还会成为你最难对付的人。通过定制一个过渡解决方案，可以让他们不必花太多时间去学习新东西，但却因此能得到极大的好处。

20.4　陷阱

即便是找到并改造其他人已有的方案，搭桥的工作也不是一件很轻松的事儿。如果你不得不构建自己的方案，那就更要知道其工作量不可小视。尤其在想到这不过是通向最终目标的一个中转站时，你可能会觉得有点没劲。不过，不要忘了，就算是一个过渡性方案，也比现在的方案好得多。

不过，还有一个风险，那就是大家会接受这个过渡方案，你再没有机会替换它。这并不一定就是一件坏事。对此，你得灵活一点，学会接受现实。这也意味着你开发的临时性方案必须作为一个独立的实体得到维护。如果对此没有心理准备，一旦遇到这样的状况，你可能就会心烦意乱。在此，我还得再说一遍，即便如此也比你现在的情况好。

20.5　小结

我不会骗你，搭桥的工作量确实很大。不过，由于你可以定制这个过渡性的方案，因此它很有可能会在这个中间步骤中发挥重要作用。你可以通过它来推动大家进步。

某种程度上讲，这个过渡方案展示出了显而易见的道理。正如我跟你说的："如果你努力工作、均衡饮食、保持锻炼，那你也可以减肥。"我其实并不是在忽悠你。如果你也尝试过用其他方法来布道，但结果不理想，不妨考虑一下这个方案。

几点建议

实际搭一座桥有点枯燥，不过你可以尝试去研究一下，去找一个已经存在的方案，将它作为你的桥。当然，也可以自己亲手去搭一些小桥。



- 看看你的目标解决方案，再看看你的团队使用的平台上是否还存在其他类似的现成方案。
- 花点时间，针对编写过渡方案构思一个计划：
    - 搞清楚你要改变团队最信奉的哪一条法则；
    - 确定自己方案中的哪些部分是最有可能得到大家认可的；
    - 估算一下搭这座桥需要花多长时间。

第21章 来点刺激

说句掏心窝子的话吧。如果你能强迫别人使用你的工具或技术，你恐怕早就那么干了。如果真能那样，你也就不会看这本书了。不过，我倒知道一种间接的方法，让你能够强迫别人使用你想让他们用的东西。你可以搞一个使用你的工具或技术的内部方案，只要这个方案够吸引人，大家自然会用上那些必需的东西。



创造某种有刺激的方案，需要结合前述的策略和技巧，某种意义上这是那些策略与技巧的综合运用。让这个方案对别人有用，你就展示了技术；通过实现这个方案，你也能取得经验；你还要注重合力，利用你的工具和技术去解决公司问题。这些都没有错，但本章的这个技巧却跟其他技巧不大一样，它明显地超越了那些技巧。它不仅仅是体现你的经验，展示你的技术，而是给人以使用相应工具的直接刺激。它也不同于寻求合力，因为寻求合力需要有待解决的商业问题，需要有管理层的强制推行。而这个技巧则相对更柔和、更优雅，大家不会受到外力的胁迫，而是发自内心地想用。

Eclipse之美

Rupesh想让自己负责Web开发的同事使用Eclipse，有以下几个理由：

- 他不喜欢别人正在使用的专有工具；
- 他打心眼儿里相信Eclipse才是更好的选择；
- 他所希望的是使推进源代码控制、构建脚本和持续集成这项工作变得简单。

这家公司的所有Web项目都有一个难受的地方：在项目中要使用公司的Web模板。倒不是说使用这些模板很困难，而是太费时间。想

要完成任何工作，都必须向几十个图片里添加项目的标志。其实，最令人讨厌的就是这些工作完全都是重复性的手工劳动，而且无论项目标志还是公司标志，只要在项目的中途改变了，同样的工作都少不了要再重复一遍。

Rupesh对使用Java和JIMI（Java Image Management Interface，Java图像管理接口）有一些经验。经过一番测试和验证之后，他写了一个程序，可以自动将标志加到图像上。结果，他并没有直接把这个程序公开出来，而是以插件的形式集成到了Eclipse中。使用这个工具，原先要花几个小时的工作，现在只要数秒钟就可以搞定了。

Rupesh把这个工具展示给团队的其他开发人员看，引起了一阵欢呼。即使是此前一直都讨厌Eclipse的人都改变了心意，就因为从今以后可以不必再一张一张地手工处理那些图片了。

这样一来，Rupesh就可以把精力放到他所希望的其他转变上了。更难得的是，让大家接受Eclipse根本没有用多少政治资本。现在，他可以利用这些资本去做更多的事儿，因为他没有动用管理层就让大家用上了Eclipse。



21.1 技巧分析

这个技巧起作用的关键在于它利用了人们内心的愿望去征服他们。没有强迫、没有唠叨，也没其他方式的推销工作，他们使用你的工具或技术，是因为这个工具能帮他们化解其他燃眉之急。人们在没有别人强迫他们放弃对某些东西的抵抗时，反而会更愿意接受这些东西。

21.2 给大家提一提神吧

要想创造一些吸引人的东西，第一步就是要找到让自己的团队最难受的问题。哪个团队都有最难受的问题。在电影《上班一条虫》（*Office Space*）中，大家经常挂在嘴边上的一个玩笑是TPS报告，这个TPS报告到底是个什么，剧情并没有交待。观众只知道剧中的主角忘了在报告中使用新的封面。但就为这，公司所有人都埋怨他把事情搞砸了。没人关心报告的内容是什么，

反正就是怪罪他“做错了”。更重要的是，如果那件事那么容易就会搞砸，想一想都让人害怕呀。

你可以找找自己身边的“TPS报告”。可能是文档，可能是费用报告，甚至可能是产品本身。总之，这件事占用了你们的时间，而且这时间不花还不行，可它又不是你们的核心工作。无论是什么事，都是你接下来所要解决的重点问题。

第二步当然就是解决该问题。能否成功解决问题需要打个问号。一般来说，存在这个讨厌的问题，可能是因为缺少一种自动化的方式来减轻大家的痛苦。然而，日复一日地，居然就没人认真地想过这个问题。每一年都会涌现很多新工具和新技术，而其中能够用来解决这个问题的工具或技术恐怕不止一种。

在写出自己的工具并做好准备之后，你必须向自己的队友们展示它。好，你现在面临一个严峻的问题：什么时候跟大家讲一讲需求？你必须得实话实说，但没有必要一上来就强推。要知道，“我现在有一个非常棒的工具，你们应该知道必须使用x才能……”与“在进入正题之前，我了解大家一直都不得不……”是有区别的。把好话说在前面，不代表你不真诚。当然也没必要捏造一些丑话，那也过于假惺惺了！



21.3　适用对象

来点刺激这个技巧对以下类型的怀疑者行之有效。

21.3.1　孤陋寡闻型

这个技巧对孤陋寡闻型的人来说提供了一个快速通道。在接触你的项目之前，他们对你最终想要推荐的工具和技术闻所未闻。只是在使用你的项目的过程中，他们才有所了解。如果你的项目做得不错，那么两者都会给他们留下好印象。

21.3.2　随波逐流型

同样，就跟对付孤陋寡闻型的人一样，这个技巧也可以让随波逐流型的

人迅速站到你这一边来。你通过自己示范性的项目解决的问题，正是他们在以往工作中不得不面对的棘手问题。你这样做，就让他们面临两个选择：要么维持痛苦的现状，要么采用你那减轻痛苦的方案。甚至，你都不必对他们进行什么引导，他们自然会慢慢地放弃抵抗。

21.3.3　时间紧迫型

既然这种技术能够攻克一个让人难受的问题，那它对时间紧迫型的人会特别有效。并不是所有问题都消耗时间，但多数情况下都是。其余的问题基本是财务限制，解决了这个问题，省下的钱就可以用来购买更多时间。

21.3.4　百般挑剔型

百般挑剔型的人之所以会挑剔，因为他们打心眼儿里就不相信会成功。而通过你的方案来演示某工具或技术可以有效解决问题，事实上已经先发制人了。也许他们不会承认你已经有效解决了问题，但却能降低他们说服别人的可能性。



21.3.5　激情燃尽型

与百般挑剔型的人类似，你的方案证明激情燃尽型的人是错的。你推荐的工具和技术完全可以发挥作用。不过跟往常一样，要记住不能攻击激情燃尽型的人，但你可以证明他们完全可以从跌倒的地方爬起来。

21.4　陷阱

首先，这个技巧建立在一大堆假设的基础之上。假设你能找到这么一个令大家都很苦恼的问题，假设你能使用自己的方案，假设你能让解决方案有吸引力，而且还要假设你能够说服大家都来使用，这样才能成功。所以，这种技巧从某种程度上与其他技巧也十分相似，那就是依赖于恰当的机会。因此，你必须得自己去寻找，不要指望天上掉馅饼了。

你能把自己的方案与某些东西联系到一块，并不意味着你的同事不能断开它们的联系。在前面那个Eclipse的故事里，你的方案就是写一堆Java代码，

然后把它做成Eclipse插件。要问一句这个插件是否可以独立出来，并不是一件很难的事儿。如果真的有人提出这个问题，那说明这个方案虽好，但还不足以帮你达成目标。

同样，仅仅是让大家使用你的工具或技术，并不能保证他们会全面使用。再回想一下那个故事，Rupesh的同事很有可能打开Eclipse只是为了使用那个插件，用完了他们就会打开其他IDE继续自己的工作。换句话说，这个方法可以给你一个机会，但却不能保证这个机会可以自动帮你达到目的。

21.5 小结

综上所述，这个技巧不能帮你把所有问题都解决掉，但它具有几个明显的优点。大家不会讨厌你推荐的工具或技术，因为与被强制、被指挥或被命令相比，使用有吸引力的工具显然更容易接受。在此期间，你还能帮自己的团队解决一个令人苦恼的问题，这多好啊。而假如你能利用好这个机会获得进一步展示的空间，那这个技术就是切实可行的，而且也将是十分有效的。



几点建议

为解决令团队成员都感到苦恼的问题，从无到有地开发一个完备的方案需要勇气。下面是几条建议，或许能帮你鼓起勇气。

- 找出令团队成员都感到痛苦的事情。（不用说，你必须得这么做。）
- 看看哪个问题可以用你的工具或技术来解决。
- 写一个方案出来，评估一下看值不值得去实现。

Part 4 第四部分

策　略

本部分内容

第 22 章

简单，但不容易

通过阅读前面那些章，你已经了解了一批方便识别的怀疑者模式，以及一套与之战斗的技巧。图22-1展示了这些技巧与适用的怀疑者模式之间的对应关系。

	孤陋寡闻型	随波逐流型	百般挑剔型	激情燃尽型	时间紧迫型	发号施令型	不可理喻型
取得经验	●	●	●	●			
展示技术	●		●		●		●
传达理念	●		●				●
建立信任			●	●			●
适当妥协					●	●	
公之于众	●		●	●		●	
注重合力					●	●	
搭一座桥		●	●	●	●		
来点刺激	●	●	●	●	●		

图 22-1　怀疑者模式与应对技巧

有了这些一一对应的怀疑者模式和应对技巧，也就有了一套可以随时查



阅使用的战术。但光有这个图还不够，你还得有效地去运用它们。为此，还需要几个策略。

克敌制胜的策略极其简单。但简单并不意味着容易。把一块圆形的石头弄到山顶很简单：只要沿着山坡把它往上滚，一直到滚到山顶就行啦。但说说容易，做起来就难了。

有了“简单但不容易”这个心理准备之后，下面就来见识见识这些重要的策略吧：

- 无视敌人；
- 先易后难；
- 借力支持者；
- 说服管理层。



就这些？就这些。你只要尽最大可能对那些完全敌对的人视而不见，别在他们身上浪费一分钟时间。你只要对自己所能接触得到的那些意愿最强烈的人采用一些战术，然后，鼓动被你说服的人们参与一两个战术的实施。这样转变一批，鼓动一批，再投身于转变下一组最容易转变的人，循序渐进，直至你把所有能够转变的都转变过来。

如果能够把所有人都转变过来，那恭喜你——你现在就可以收工了。如果没有，那恐怕就需要评估一下。站在你这一边的人是否已经足够多，多到能让人觉得你的工具或技术值得考虑？如果是，那也算是可以收工了。如果不是，那最终还得把管理层拉过来。如果一切顺利，管理层就会强制推行你的工具和技术了。

简单，对吧？不过，我刚刚是不是还跟你说过——不容易？

第 23 章

无视敌人

什么？躲着点那些不可理喻的人？乍一听，这似乎有点不成熟。在自己努力的过程中，想要排除一部分人——即便是不可理喻型的人，听起来有点消极。最终，你不还得面对这些人吗？

不。

噢，抱歉，你想得到一个更深入的回答？



好吧，不可理喻型的人，他们质疑你并不是因为有什么合理的理由。他们的动机难以捉摸，也经常搞不清楚。正因为如此，若想先找到他们的动机再想办法解决，需要花费的努力会超出想象。一般来说，还要根据每个人的具体情况给他们量身定制方案。光靠给他们做一次演示，说明你对相应的工具很在行，是专家，那还不够。你必须要证明他们认为的那些不合理的甚至是一些不可能的问题，跟你推荐的这种专业开发工具扯不上半点关系。

还有一种可能，就是即便你找到了他们反对的真正动机，也是束手无策。也许他们个人对你、你的老板或者你的部门就心存不满。你不管怎么努力，他都不会听你的。如果想让他们在这些问题上有所让步，没有几个月甚至几年的工夫根本不可能。

以上分析的结论是：投入高，产出低。犯得着吗？还有很多其他类型的怀疑者需要你去争取啊。对这些人，真的没有必要浪费时间和精力。

23.1　怎么无视敌人

既然如此，是不是要着意回避一下这些人，让他们觉得面子上还过得

去？不必。我的意思很明白，不要向他们推荐，不要想说服他们，不要在他们身上浪费时间。

但这并不是说不能跟他们交流。如果在你安排好的某个活动中，比如在团队开会时，你正跟大家讨论某开发工具，而他们也加入了讨论，继续——让他们跟着一块儿讨论。在你应用各种技巧的同时，也让他们了解一些相关信息。

在此期间，要注意不要被他们牵着鼻子走，不要让他们抢走其他听众的注意力。他们提问，你就回答，然后继续自己的思路。如果他们跟你纠缠不休，你就有礼貌地向他们说明，还有其他要点没有讲完，而你非常愿意会后再找他们讨论。

23.2　为什么不好处理



冲突、分歧，还有其他争论可能会带来无尽的成果。但是，这些也会激发敌意和怒气。双方争吵的时间越长，特别是双方谁都寸步不让，怒气和敌意就会越积越多。对于不可理喻型的人来说，怀有敌意和怒气简直是必然的。

持久的敌意会导致我们把对手视为敌人。敌人是必须要消灭的——这是人类的本能。于是，为了消灭他们，我们就会接近他们。而接近不可理喻型的人，只能使结果与我们的初衷背道而驰。

关键在于，要把不可理喻型的人想象成障碍物，而不是敌人。敌人必须要打败，而障碍物只要绕过去就行了。绕过障碍物就简单多了，谁平时走路不会绕过障碍物呢？

综上所述，招惹不可理喻型的人，结果只能白白地浪费宝贵的时间和精力。但他们会不依不饶地找上门来跟你对阵，让你情不自禁地要跟他们决斗，这样就免不了跟他们绑到一块儿了。因此，躲避他们要像躲避扎人的蒺藜一样，更不要主动去找他们，不在混战中作无谓的牺牲。这样，再想象一下“无视敌人”是不是就容易理解了？这不就是我一开始说的嘛，你们非得要一个更深入的回答！

第24章 先易后难

现在，你对应该如何对待不可理喻型的人已经有了结论。接下来，你需要开始自己的转变工作了。从哪些人开始呢？简单啊，哪些人最容易转变，就从哪些人开始。

你的想法可能正好相反——先转变最难转变的，其他人会紧随其后。但在现实中，这是行不通的。道理太简单了：在这种情况下，大家会说，谁都不这样做，为什么我还要这样做呢？

但正如第25章将会讨论的，被转换之后的人，将成为你整个过程中最强有力的同盟。转换的人越多，你的日子就越好过。当然，这里说的只是数量，不是质量。至于说Bob的编程水平是不是比Ed和Steve俩人加起来都高，那无关紧要。只要Ed和Steve更容易转变，就先从他们下手，然后获得他们的支持，再一块去转变Bob。



24.1 难度分组

根据我的经验，可以把所有类型的怀疑者划分成三个难度组：容易、难和最难。为什么没有一个不易不难组？好，根据我的经验，第一组和第二组之间的难度相差确实很大。在转变完第一组之后，开始转变第二组之前，必须做好足够的克服困难的心理准备。认识到这一点非常重要，因为如果没有充分考虑这两组间的难度差别之大，没有做好准备，那对第二组的转变工作很可能会导致你折戟沉沙。

24.2 容易

对任何人我都不敢说容易。说这一组开发人员容易转变，并不是说他们很容易被说服。这里所说的容易，是指对他们你知道该做些什么。根据他们的类型，你可以为他们提供信息，可以引导他们。就像你在山坡上推动一块静止的石头，然后他们就可以滚动起来。

24.2.1 孤陋寡闻型

孤陋寡闻型的人是你要过的第一关。你能左右他们对一种技术的第一印象。你对这种技术的介绍，会成为他们认识这种技术的对比和参照。这对你来说是一个巨大的优势。

想想你的亲戚给你转发的每一封有关某某传闻的邮件，你想过要反驳他们吗？那几乎是不可能发生的。即使有Snopes[①]之类的专事辟谣的站点，人们也听不进去。为什么？因为正当他们信以为真的时候，你跑去揭露真相了；这时候他们怎么会听得进去呢！给人的大脑里植入一个观点，比改变已有的观点要容易得多。



我在这里不是想说大家会像相信自己亲戚转发的邮件一样随便去相信任何事。只是想说人的第一印象是真实存在的，而且很难改变。所以，你应该好好利用这一点，做第一个诠释自己的工具和技术的人。

24.2.2 随波逐流型

随波逐流型的人基本上就是想找一个人来领导他们。要把他们拉到你这边来，你就得具备这种领导力。说到这里，但愿你不会天真地以为，领导他们就是跟他们说："用这个吧。"

---

① Snopes.com是美国一家专门核查并揭穿谣言和传闻的网站，创办者是一对美国中年夫妇。针对各种传闻，网站会验证各种说法的真实性，并且毫不留情地揭穿那些彻头彻尾的谎言，网站会用"真/假/不确定"的可信度评定。（简介来源：http://t.cn/aYfRuu。）
——译者注

真正的领导意味着为寻求领导的人提供持续不断的支持和引导。如果他们遇到了麻烦事，你就得出来解决。你还得给他们提供足够的理由，让他们能够一直使用你的工具。

24.3 难

如果说转变第一组的人就像在山坡上向下推动某个静止的物体，那么转变这个难度组的人，则无异于要改变已经向下滚动的物体的方向。这就难多了。为此，必须克服这些人的惯性。换句话说，推动他们并不是只克服重力就行的，还要克服他们的反作用力。

更不容易的是，不可能通过一件事就把他们转变过来。一次演示、一次项目分享，或者一次开发成功，都不足以说服他们。只有多方面的成功才能让他们改变心意。而多方面的成功都包括哪些方面？这个结论又因人而异。所以说，难度级别又提升了。不过，这些人还是有可能被说服的。



24.3.1 激情燃尽型

在这个难度级别里，激情燃尽型的人是相对最容易转变的。应该说，他们曾经觉得采用你的工具或技术是个不错的选择，只不过不是现在。这个结果不错：他们是能够接受你的工具或技术的。你只是要重新燃起他们的希望之火。然而，从另一个角度看，激情燃尽型的人也不一定最容易转变，因为他们可能是所有怀疑者中最有激情的。在他们决定再试一试之前，他们的反对很可能会非常激烈。

24.3.2 时间紧迫型

对于时间紧迫型的人来说，最重要的是他们对你的工具或技术虽然反对，但不会有明确的意见。他们反对的不是你的工具，而是对当前的做事方式的改变——他们无法忍受这种改变带来的时间成本。

如果不能证明你的工具最终能够节省他们的时间，时间紧迫型的人不会给你机会。之所以把他们的问题放到稍晚一些时候来解决，就是因为需要提供一些验证。当随波逐流型的人、孤陋寡闻型的人，还有激情燃尽型的人纷

纷站到你这一边时，你才可能给他们提供来自第三方的验证，证明你的工具可以为他们节省时间。虽然这种验证不一定一下子就能把他们转变过来，但却能够提高这种可能性。

24.3.3 百般挑剔型

百般挑剔型的人好像经历过很多失败。对你的每个论调或观点，他都能摆出一些统计数字，或讲出一个反面故事。看起来，说服他们似乎是一件不可能的事。可是，他们有一个非常大的弱点：想让别人认为他们聪明。因此如果有其他聪明人已经接受了某种技术，他们会在自己也是一个聪明人的迫切渴望下也加入你的阵营。但要达成这个效果，你必须让一些聪明人先接受你的技术，这也是为什么把这种类型的人放在最后一位的原因。



24.4 最难

对于最难的一组，除非你能把其他组的一些人拉过来，否则是不可能说服他们的。对于容易的一组，如果你周围不存在那样的人，你可以跳过。但对于最难的一组，如果没有人站在你这一边的话，根本没办法去跟他们说。

发号施令型

管理人员是最难说服的。跟他们沟通需要付出很多努力，必须使用特殊的说辞，尤其是要使用他们能听明白的术语。此外，管理人员并不是只为你自己配备的。每个人都希望占用他们的时间，得到他们的关注。他们不仅要面对你所关心的问题，更要处理各种各样管理上的琐事。因此，你不仅要跟他们提这件事，更重要的是还要让他们上心。

让说服工作更加困难的是，多数情况下你不可能单独达成这个目标。你需要已经转变的人向管理者证明你的主意有可取之处。如果你无法说服那些至少表面上是你队友的同事，那凭什么让老板相信你？他们持这种态度没有什么错。在什么情况下有可能出现这个问题？你并没有跟自己的队友提及你的工具，而是直接就跟管理人员谈。或者，你曾尝试说服他们，但没有人听，所以你才找到管理人员，希望他们能帮你强制推行。结果是显而易见的，如

果你在没有取得同事支持的情况下跟老板谈，那就是因为你根本没有得到支持，而不是因为想为自己的发现保密。

好了，以上就是一个大致的难度划分。这样划分有它的合理性。但你也不要太教条，如果你觉得自己可以跳出这个顺序，先说服这里提到的更难说服的怀疑者，不要犹豫。总之，我的意思是先说服那些最容易说服的人。假如条件允许，你有可能先说服百般挑剔的人或者管理人员，也绝对不要放过这样的机会。



第 25 章

借力支持者

此时此刻，你应该像躲避瘟疫一样躲着不可理喻的人，应该已经让一些人接受了你的技术。但是，仍然有人质疑你。你认为可以说服他们，但他们还没有听从你的建议。

他们没有因为你的努力而动摇的原因有很多。有些人可能还记得，在上班第一天的时候，你无意中敲错的字母导致1000个账号被删除。有些人可能只愿意听从某些人的建议，无论他们说什么，这些人就是愿意听。



此时此刻，你需要一些帮助。你需要那些已经转变的人来帮你一把，把这件事做成。你需要他们去说服其他怀疑者，需要他们把你的解决方案变得触手可及。

要让他们出面帮你，必须向他们提供一些资源，让他们有能力加入讨论。你需要他们去影响各自能够影响的人。最后，你需要他们的声音淹没那些反对的声音。

25.1　请求帮助

想要得到已经转变的人的帮助，就必须跟他们说出来。如果你想让他们积极地参与到你的努力当中，或者只是想让他们重复表达他们曾经表达过的意见，就必须向他们说明白。在未经他们允许的情况下，如果让已经转变的人发现你在跟别人谈起他们，或者转述他们曾经说过的话，那你可能会丢掉这个同盟者，会让人觉得你这个人不可靠。

在提出要求的时候，必须让他们知道你在做什么，比如，“我想让公司

通过[X]做到标准化，你们已经认可了[X]，我现在希望得到你们的帮助——能跟我一起努力吗？”虽然这个例子本身有点傻，但相信你已经领会了其中的意思，那就是——毫无保留。

有时候人们会犹豫不决。找他们解决技术问题是一回事，但找他们在公司里推广一个工具或技术则是另外一回事。不过，别忘了你是想让自己的工作环境变得更好。你并不是为了在公司抬高自己的地位才那么做的；你只希望得到一个结果。只要你能把握住这一点，那你的同盟者也可以。

假如你对已经转变的人有任何不诚实，那你就有操纵他们而不是与他们并肩战斗的嫌疑。操纵他们，就是利用他们。只有真诚地向他们请求帮助，他们才能与你并肩战斗。

除了利用人这个问题之外，还有一个问题：看似利用人。后者比利用人更难处理。你可以控制自己是不是对大家真诚，但你控制不了别人认为你在利用人。提前做到以诚相待和毫无保留，通常都可以为你正名。



所以，无论什么情况下，都要做到开诚布公，包括解释你的目的和请求帮助。有人会帮你，有人不会。跟愿意帮你的人同心协力，但也要尊重那些没有帮你的人。然后把精力都用在转变其他同事上。

25.2 创造布道者

他们已经同意帮你了，接下来你得先帮助他们，让他们知道怎么帮你。你需要用工具来武装他们，让他们在面对怀疑者时能够学以致用。同样，毫无保留是重点。

首先，要让他们明白为什么让更多团队使用你的工具或技术很重要。“我们在维护那些纠缠不清的陈旧代码上花的时间太多。如果我们使用框架[X]做到标准化，那维护时间就可以大大减少，而我们可以更加专注于……”就要这么简洁明确；要让大家知道是跟什么在斗争。

其次，明明白白地告诉他们要面对哪些怀疑者。让他们知道谁和谁为什么会质疑。让他们了解谁属于激情燃尽型，谁又是时间紧迫型。

然后，让他们知道你的整个战斗计划。让他们知道你正针对哪些人，想按什么顺序做，以及这样做的原因。让他们知道你对成功的构想，以及什么时候可以宣告这场战役结束。

本书不保密

没有必要对归顺你的人隐瞒本书介绍的内容、战术以及策略。本书的理论体系自身什么也做不了，它只能帮你做事更有条理。真正管用的是你的东西、观点和实例。这个体系的成功并不建立在向他们隐瞒什么的基础上。因此，既然说了毫无保留，那就不隐瞒。

最后，把他们派出去。让他们帮你说服怀疑者，试试让他们去改变某些想法吧。

25.3 交叉推广



在你的布道者为你出去工作的时候，你得为他们工作。你必须到处宣传他们的成就。跟同事或领导聊一聊，说说他们使用你的工具或技术之后有哪些成果。不过，在谈到他们的先进事迹的时候，不要说他们的成功得益于你推荐的工具或技术。只管宣传你的布道者们。只需尽力宣传他们，而不是宣传你的方案。一旦有人感兴趣了，刨根问底起来，再在介绍的过程中提一提你的方案。

同样，也跟你的支持者们谈一谈这些。让他们在提到你的技术时也要注意类似的要点。要求他们先提你的方案，然后等谈得深入的时候再提及他们的成功。

为什么强调大家都先宣传别人呢？因为人们很少愿意听别人自吹自擂，而更愿意听到你谈到他人的成功。我们对那些厚脸皮的自我夸耀已经麻木不仁，不愿理会了。对类似的话，我们只会充耳不闻。因此，不能宣传自己，要宣传别人。然后，让听众的好奇心把他们带到你的面前来。当然不会每个人都找上门来，那是正常的。而一旦真的发生了这种事儿，那效果就不是一般地好了，而你们的努力也就值得了。

交叉推广是一种创造机会的方法。你必须学会等待，等到时机成熟再谈转变的事儿。到时候，别人会反过来请你多介绍一些信息。但你可以自己再创造机会："你以前听说过Beta组吗？"而大多数情况下，当你们讨论某个成功故事的时候，多数人都希望知道是怎么成功的。

25.4 消耗关注

一家公司或组织中的关注是一种零和博弈。人们的注意力是有限的，得到一些人的注意，就意味着其他人会失去他们的注意。这就是说，通过抓住人们的注意力，不仅你和你的支持者能得到更多的帮助，而且能够减少大家对怀疑者的关注，从而降低了说服他们的难度。

赢得注意力的多寡因公司或组织而定。你们内部有没有博客或者留言板什么的？如果有的话，你可以和你的支持者们在那里讨论你的工具或技术。你的同事是不是都在使用公开或私密的社会化网络工具？如果是，那你们得到那里面去叫喊。当你在电梯里遇到某个领导时，是不是从来都不知道跟他们谈点什么？那好，就在那里展开交叉推广吧。



总之，只要是在合适的地方，只要能够展示一下自己，或者能够谈谈支持者的成就，那就放开了去做。如果出现了一个非常自然的机会，能够让你这么做，不要犹豫。要尽量创造这么一个环境，即对怀疑者的每一条说辞，你这一方都有好几条反驳理由等着，让它淹没在正面的声音中。

借力支持者是你的一个强大策略。一个人势单力孤，很容易被忽略，但通过让每个人都重复你的说法，就没有那么容易被忽略了。然而，这个策略也要经受道德的考验。没错，你必须交叉推广，从怀疑者中脱颖而出，但也要小心，不能忽视第8章介绍过的信任的重要性。

第 26 章

说服管理层

有些工具和技术要求百分之百的“兼容”。并不是任何时候都可以说服所有人，而且你也知道不可理喻的人就没有办法说服。因此，整个计划的最后一步，就是去说服管理层，或者让他们出面，或者通过制度强制推行，总之要让所有人都使用你的工具或技术。通常只有管理层的强制命令，才能让那些自命不凡者接受现实。



26.1　希望管理层做什么

简单说，你希望管理层出台一条规定，规定在特定的场合下必须使用你推荐的工具和技术。然后一切就简单多了，那些曾经质疑你的人，现在都会按照你的想法去完成一部分工作。

强制执行的方式有很多种。通常，人们被告知必须做什么。不照做就要说出理由。视工作或技术以及组织的不同，员工每年的工作总结通常都要把执行公司制度的情况作为一个内容来写。

负责把关的人也可能推动强制执行。这些人掌握着怀疑者们平常工作或者把工作成果推向客户所必须的资源。举个例子，如果你需要服务器管理员把你的代码放到产品服务器上，那他们在发布你的代码之前，就可以检查你是否执行了公司的制度。

26.2　怎么做到

说服管理层的难度也不一样，这可能取决于作出决定需要履行的程序，或者你的团队所拥有的灵活度。先不考虑这些因素，若要说服管理人员，必

须让他们明白这件事必须要做，你也不是唯一一个希望这样做的人，而且，你对为什么这样做能给出一个让他们信服的理由。

26.2.1　解决管理问题

第10章也说过，你得想办法解决管理人员的问题，而非你自己的问题。向管理人员展示时，原本的技术问题，现在必须转为管理问题。

从技术问题转变到管理问题并不困难，只要你训练一下自己的管理思维方式。开发人员时间的浪费，不就是公司金钱的浪费嘛。性能低下最终还不是导致硬件资源的浪费？不安全的环境在你们这个行业里可能就意味着要造成财产损失，或者带来法律责任。

在此基础上，还需要更进一步。只说某个改变会节省开发时间还不够。你必须计算出来到底能节省多少时间。有了时间，就可以再根据薪资水平或估计费用来计算出节省的成本。然后，把这个结果拿给管理人员看。这才是用他们的语言，说他们的话，解决他们的问题。



26.2.2　用数字说话

你不是单兵作战。（你已经做完了前面几步，已经有了不少支持者了，对吧？）而且，你也不能让管理人员觉得就你一个人来请命。要让他们知道，你现在代表的是一个团队，而不是怀着私心来给他制造麻烦。

即便你是正确的，而他们也相信你是正确的，仅仅是你一个人也不足以说服他们。只听一个人的说法，然后就发布命令是有风险的。如果你给他们的感觉是你一个人在对付很多人，那结果很可能是不。如果反过来，你代表大多数人，那你的想法就很值得考虑了。

26.2.3　解释为什么需要强制执行

多数情况下，当你把一个诸如此类的技术问题抛给他们时，他们不会感到格外吃惊。你得承认一个现实——你在抱怨某个地方出了问题。因为抱怨，所以你才会想改变。结果就是给管理者制造麻烦，需要管理人员出面摆平。

他们有必要知道，为什么自己要管这件事。

正如前面我们说过的，答案就是开诚布公，毫无保留。把你的行动过程告诉他们，把你努力说服同事以及联络了支持者的事告诉他们。让他们知道，在请求他们出面之前，你已经用完了自己的资源。

26.3　接下来怎么办

运气好的话，管理人员会同意你的说法，发布一条命令。问题迎刃而解了，对吗？

没有这么简单。仅仅是下命令执行，不等于就真的执行了。你必须得监控、跟踪，最终报告执行情况。为什么是你？因为整件事从一开始就是由你规划的。没有监督，命令也白搭。当然，我这里的意思不是让你在自己的同事中潜伏起来，但你确实需要鼓励大家去遵从。



说到底，请求管理层有时候也不是必需的。可能使用前面三个方法就能说服别人了。况且，也许你的方案也不必追求百分之百的“纯粹性”，而你只要潇洒地跟那些不合作的家伙挥一挥手即可。

要注意的是，把管理人员搬出来的做法在这场冲突中是一颗“原子弹”。没错，它可能威力无比，能够帮你实现目标。但在你使用了这一招之后，人们很可能就不再在意你，何况这样做也不保证一定能成功。说服别人总比强迫别人更好，但你也不总是有机会走这个捷径。

第 27 章

最后的话

你已经做到了。你忽略了不可理喻的人，把怀疑者分成了几组各个击破，实现了转变。你和支持者们，要么自己说服了所有人都使用你的工具或技术，要么说服了管理层强制实施。你的方案被采用了，你的任务完成了。就这样了吗？当然不是。

27.1　经验教训

即便是你成功之后，也还是有陷阱。人类的本性会让你以为到了某个过程结束，一定就是庆祝、和平和光明。可悲的是，事实并非如此。以下几个例子或许能够让人在成功之后冷静一下。

27.1.1　成功过度

再说存储过程

几年前，我在一个家Web开发公司工作，这个公司有个问题：Web应用服务器时不时地就会崩溃。经过追查，发现问题出在应用服务器与数据库关联的代码中。

评审之后，原因清晰地浮出水面，是个别写得不够完善的数据库操作代码在作怪。我们这个Web开发团队的成员在编写客户端代码和业务逻辑方面都非常出色，但大家都不是DBA。解决方案就是那条可怕的存储过程法则：所有SQL语句都必须放在存储过程中，而且这些存储过程在放到产品中之前，必须全部经过DBA的审查。

这是我在推广技术的生涯中取得的第一个成功。DBA说服了很多



开发人员接受这个方案。其他人也被管理人员强迫服从了。

这个方案解决了问题。数据库操作导致的服务器崩溃再也没有发生过。应用服务器比以前稳定多了。

时光如梭，转眼5年过去了。有一次我跟那家Web开发公司的一名新员工一起喝饮料。

“我觉得很没劲。我希望让大家使用ORM框架，但所有框架都要求直接编写SQL。DBA不允许这么做。不管怎么说，我想使用的框架编写的SQL比我写的都好。我现在恨不得杀了那个制定这条规定的人。”

我悻悻地转移了话题，手摸索着找到了饮料罐的拉环。

这个问题说明，现在的DBA只知道强迫大家编写存储过程，但并不知道这样做的原因。这个技术和规定还在延续，但当初的问题已经不存在了。



没错，一旦你推广了某种技术，可能会令人无法摆脱。随着技术的变迁，原来的技术越是陈旧，要摆脱它们就越困难。

在前面的例子中，我犯了一个错误。我把技术和规定推给大家，但却没有告诉大家原因。如果大家都知道了这样做的原因，那DBA会不会放轻松一点儿呢？可能吧，谁知道呢。

一定要解释清楚为什么要那么做，而不是简单的怎么做。

27.1.2 莫求回报

终于……

我离开上一家公司，感觉自己像个失败者。我曾经推广过框架、代码生成以及单元测试，但全都无功而返。时间紧迫型的人质疑这些技术的陈词滥调把我的耳朵都磨起茧子来了。经过了几年的努力，我发现无法改变他们。

同样，时光荏苒，一晃几年又过去了。有一次，我遇到了一位前同事。他到那家公司是我推荐的，他是一个非常出色的程序员。之所以觉得他很出色，部分因为他对我推广的每个工具或技术都积极响应。

他跟我抱怨自己的团队：

"总之，他们根本不喜欢这个框架，因为它有点复杂。但他们也不喜欢另一个，因为用那个来开发会导致开发周期变长。结果，差不多5个月前，我们转向用第三个，因为它支持……"

让我把话说明白点。那些曾拒绝考虑使用框架的人，在争论了哪一个好之后，最终还是采用了一个框架。对了，还有呢，他们也使用了一个单元测试框架，自动生成初始的单元测试，再自己编写其他测试。

也就是说，无论我当初怎么努力，这个团队就是不接受，至少我在那儿的时候如此。在我离开那个团队两年多以后，他们终于有所改变了。

或许是因为我做了很多前期的铺垫工作，或许是我已经软化了某些人，他们没有赞同我的意见，但随之赞同了新来的人。也有可能对那个团队而言，只有慢慢地改变才是唯一可行的。不管怎么样，改变是改变了，只是慢了一点罢了。



27.1.3　你可能错了

正确的答案，错误的答案

这又是另一种成功故事。我让一个没有使用过源代码控制的开发团队使用了Subversion。他们完成迁移之后，晚上都睡得很安稳，不需要再担心一个错误的`rm *`将会导致整个站点消失。

可是还有一个问题。我一点儿也不知道怎么配置SVN服务器。我基于当时的代码库结构设计的架构实在太可怕了。在头几个项目上，它没有问题。但当项目数达到两位数的时候，性能出现了严重问题，所有检出都被拒绝，这个系统也变得无人问津。

我耽误了进度，把事情搞砸了。结果还创造了一些激情燃尽型的怀疑者。

没有人是完美无缺的，也没有人能预见未来。这就是所谓的不可预见性。经验当然有用，但你同样有可能把事情做错。做错了，就接受它吧。

更加重要的是，你有了自己失败的经验。承认这些经验吧。这些经验也值得你宣传。更重要的是，别忘了解释清楚失败的原因。

在前面的例子中，Subversion本身不是问题，问题在我。我承认了，然后跟大家宣讲，解释。正因为如此，我获得了时间，可以迁移到另一个Subversion的解决方案。这次成功了，我没有让大家的激情继续被伤害——这完全是因为我在解决问题之前，已经失败过一次。

27.2 成功之道

在取得了一些成功之后，有时候你可能会忽视一个事实，那就是你的有些同事依然是怀疑者。没错，他们接受了你推荐的工具或技术，但当你再推荐其他东西的时候，他们还是会质疑。

事实上，面对我们这个行业日新月异的变化，面对成百上千的新工具和新技术，我们每个人都可以说是孤陋寡闻的怀疑者，就因为我们没有听说过这些工具和技术。



这里所要强调的是，每一次都必须重新开始。多次成功可能会增加你的可信度，但除此之外也没有什么值得夸耀的。说起来确实很让人气馁，但这就是布道者的宿命。

在一次次的尝试中，我总结了几条经验，可能会对你有所帮助。

- 已经被你打上随波逐流、百般挑剔和时间紧迫等标签的人，下一次很可能还是同一种类型；
- 孤陋寡闻的人，下一次更有可能还是孤陋寡闻；
- 发号施令的人，当然还是领导；
- 不可理喻的人也可能会漫无目的地瞎掺和，不过如果他们就盯上了你，而非技术，那他们很可能还会故伎重演。

27.3 问题会扩散

我们经常会有一种错觉，似乎在人们采用了我们的工具或技术之后，我们解决了一系列的问题。按照这种思维模式，问题是有限的，我们可以减少问题。不幸的是，问题是无穷无尽的，它们就像空气，而不是水。换句话说，它们总是试图扩散，并占据尽可能大的空间。

节省了一大堆时间，就不必再花钱买那多么“人月”了，因而就可能会有人丢了工作。解决了一个问题，就可能会牵扯出另外一个问题。无论如何，不管你成功多少次，这个世界也不会变得完美无缺。

27.4　只有过程，没有终点

看过前面几节的内容，可能会让人觉得有点泄气。我们可以得出一个结论：即便能让大家采纳你的方案，其实也不重要。好事可能变成坏事，不好的工具可能会变得难以撼动。即使你想办法说服别人，也只是一切从零开始，更多的问题还会接踵而至。

这个结论只在一种情况下成立，那就是你无法改变自己对工作环境的认知模式。“更好”，其实不是一个目的地，而是一个方向。同样，对你技术方案的接受也不是一个终点，而是一个过程。在当前的位置和将来的目标之间，可能有很多相当不错的地方。你只需关注离开现在的位置，而不要关心去向何方。



好吧，就这些了。现在就走出去，努力让自己的工作环境变得更好吧。这是可以做到的，而且有人已经做到了。你只要加入这支以布道作为自己使命的队伍就好了。

参考文献

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.

[Mas06] Mike Mason. *Pragmatic Version Control Using Subversion. The Pragmatic Programmers*, LLC, Raleigh, NC, and Dallas, TX, second edition, 2006.

[RTH08] Sam Ruby, David Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

[Rud07] Jason Rudolph. *Getting Started with Grails*. InfoQ, 2007.

[Swi08] Travis Swicegood. *Pragmatic Version Control using Git*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2008.

[TH03] David Thomas and Andrew Hunt. *Pragmatic Version Control Using CVS*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.

布道之道

引领团队拥抱技术创新

在IT行业打拼，每个人都可能遇到这种情况：你用了一种新技术或新工具，工作效率倍增，于是迫不及待地想让自己的同事和团队都赶紧试一试。但刚一提出这件事，就有很多人抵制。如何看透怀疑者的心理？如何说服别人接受你的提议？这就是本书要告诉你的。

书中将怀疑新技术的人分成了七种类型，即孤陋寡闻型、随波逐流型、百般挑剔型、激情燃尽型、时间紧迫型、发号施令型和不可理喻型，用寥寥几笔就勾画出了每种类型人的特征，点明其心理，进而给出有效的应对技巧和说服策略，让你在团队中推广新技术时无往不胜。

努力让自己的工作环境变得更好是每个人的目标。从这个意义上说，所有人或多或少都是一名布道者。来吧，听听Adobe资深布道师Terrence的建议，你一定会有相见恨晚的感觉！

Terrence Ryan　Adobe公司资深技术布道师，致力于宣传和推广ColdFusion、Flex、Flash和AIR等技术。他毕业于宾夕法尼亚大学，曾在沃顿商学院供职十年。多年来，他始终在监理软件项目、组织代码评审、推广产业标准，不遗余力地说服同事拥抱技术创新成果。

- Adobe布道师经验总结
- 剖析七类人群心理，提供应对策略
- 助你的技术生涯步步成功

The Pragmatic Programmers

图灵社区：www.ituring.com.cn
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议　计算机/IT人文

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-26727-6

定价：29.00元